Generic Access to Symbolic Computing Services

Alexandru CÂRSTEA

Submitted in fulfilment of the requirements
of the degree of Doctor of Philosophy
at Heriot-Watt University
in the School of Mathematical and Computer Sciences
30 May 2012

Abstract

Symbolic computation is one of the computational domains that requires large computational resources. Computer Algebra Systems (CAS), the main tools used for symbolic computations, are mainly designed to be used as software tools installed on standalone machines that do not provide the required resources for solving large symbolic computation problems. In order to support symbolic computations an infrastructure built upon massively distributed computational environments must be developed.

Building an infrastructure for symbolic computations requires a thorough analysis of the most important requirements raised by the symbolic computation world and must be built based on the most suitable architectural styles and technologies. The architecture that we propose is composed of several main components: the Computer Algebra System (CAS) Server that exposes the functionality implemented by one or more supporting CASs through generic interfaces of Grid Services; the Architecture for Grid Symbolic Services Orchestration (AGSSO) Server that allows seamless composition of CAS Server capabilities; and client side libraries to assist the users in describing workflows for symbolic computations directly within the CAS environment. We have also designed and developed a framework for automatic data management of mathematical content that relies on OpenMath encoding.

To support the validation and fine tuning of the system we have developed a simulation platform that mimics the environment on which the architecture is deployed.

Acknowledgements

The work presented in this thesis would not have been possible without the contribution and support of my colleagues, my friends and family. The research results presented in this thesis are part of my contribution to the European Project FP6-2005-Infrastructure SCIEnce (RII3-CT-2005-026133).

Firstly, I am deeply indebted to my supervisors *Dr. Dana PETCU* and *Dr. Phil TRINDER* for their support and the patience they have shown. I thank *Dr. Dana PETCU* for giving me the opportunity to work in a highly professional environment and for being my mentor. I thank *Dr. Phil TRINDER* for giving me the confidence to follow my ideas and for his gentle way of supporting me throughout the whole process of developing this thesis.

Secondly, I thank my colleagues and friends from e-Austria Research Institute. I thank *Dr. Georgiana MACARIU* and *Dr. Marc FRÎNCU* for their valuable contribution, advice and commitment they have shown. I thank *Dr. Cosmin BONCHIS*, *Dr. Gabriel ISTRATE*, *Marian NEAGUL* and *Silviu PANICA* for their friendship. I am also grateful to all my colleagues from the SCIEnce project, for the long and informative discussions we had.

Last but not least I thank my family and especially my beloved *Cristina* for their moral support and understanding. It was them who gave me the power and strength to carry on in the most difficult moments.

The LaTeX template used to format this thesis was provided by my colleague from Heriot-Watt University *Dr. Lu FAN*.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Symbolic computation or computer algebra is a research domain that studies automated manipulation of mathematical formulae and equations. As described in [107], computer algebra makes possible computations in algebraic structures such as groups, number fields, Lie algebras or rings of differential operators. Using symbolic parameters during manipulations of mathematical objects makes possible generic treatment of classes of problems. Evaluation of mathematical formulae to which symbolic computation algorithms are applied is more precise since numerical substitutions are applied to irreducible terms, eliminating thus rounding errors.

Symbolic computation software systems are vital tools in several areas of modern academic and commercial research. Due to the nature of symbolic computation, large problems in this research field can not be solved using the computing power of a single computer and therefore there is an immediate need for computing infrastructures that can provide more processing power and storage capabilities. One solution proven to work for other research domains is to build collaborative computing environments based

on already existing processing power offered by computer clusters and even ordinary workstations.

Programming models and tools have evolved to provide collaborative environments under the generic term of distributed computing. Any computing system in which autonomous processing units can be interconnected through a network so they can collaborate to serve a common goal is generically identified as a distributed system. Distributed computing architectures and related technologies have evolved over time in strong relation with the evolution of hardware capabilities such as computing power, storage and communication capabilities. This evolution created new opportunities to respond better to requests formulated by both research and industry.

The main software systems used for symbolic computations are Computer Algebra Systems (CASs), and amongst them GAP [3], Maple [10] and Mathematica [24] being the most well known CASs currently used. Even if processing power and memory required to solve large symbolic problems is critical, the vast majority of the CASs were initially build to support calculations done by researchers, otherwise done by pen and paper. The initial design of these systems, the high level of knowledge and the huge effort required to adapt these systems to newer technologies were the most important impediments in aligning these systems to the latest advances in distributed computing. Amongst other requirements, interoperability with other similar systems and better support to be offered to end users for solving symbolic problems were mentioned more that three decades ago [49].

The main goal this thesis is to present a novel software architecture for symbolic computations and demonstrate its capabilities to support fundamental requirements of computer algebra specialists. The resulting infrastructure should provide required support for solving large symbolic computations. Due to its design, we demonstrate that it is versatile enough to permit easy adoption of new technologies and various CASs can be integrated as part of the architecture with a minimum effort. Within this architecture CASs play an important role because they are the actual provider of symbolic computation capabilities.

Additional components of the architecture will provide the support features that enables us to integrate and orchestrate symbolic computations engines for execution of symbolic computation workflows.

A successful distributed computation infrastructure can only be created if the requirements for the research domain are carefully analysed and a thorough investigation over the best distributed technologies to be used for such an environment are identified. The high variety of CASs and their capabilities imposes that interoperability standards for data encoding, communication interfaces and execution management capabilities are created.

The high interest in creating a collaborative environment for symbolic computations based on latest distributed models can also be demonstrated by the high number of research initiatives and joint research projects with this main goal. Among some important research projects of the last years to investigate how distributed models can be used in context of symbolic computing are "Mathematics on the Net" (MONET) [11], MathBroker [2], "Grid Enabled Numerical and Symbolic Services" (GENSS) [7] and "Symbolic Computation Infrastructure in Europe" (SCIEnce) [19].

The MONET project's aim was to develop a set of Web Services to expose symbolic computation services. These services are described using a custom XML language Mathematical Service Description Language(MSDL) [51] that describes services in terms of preconditions and effects. Based on their description, automated discovery of the correct service to solve a certain problem should be possible. Using the ideas formulated by the MONET project, within the MathBroker project a mathematical service broker was developed.

Similar ideas formulated in the MONET project were also considered by the GENSS project. Its aim was to combine the functionality offered by both Web and Grid services. The two main research directions of the project are related to discovery problems and implementing ontologies for symbolic problems.

Maple [10] and Mathematica [24] are two of the most important commercial CAS systems. They have recognized the benefits that distributed computing may bring to symbolic computations and therefore they have provided mechanisms to interconnect their systems with similar instances and even third party systems through proprietary connectors. The Grid Computing Toolbox [6] allows multiple Maple instances installed on a Local Area Network (LAN) to combine the computing power of the machines onto which they run. A similar functionality is provided also by gridMathematica [8]. Apart from the main initiatives described above there are also smaller projects and research initiatives such as Maple2G [151] which integrates Maple instances with Grid architecture and JavaMath [170] that proposes a model to expose CASs using Java specific distributed technologies. A review of current Grid-based systems for symbolic computation can be found in [152].

All the initiatives described above demonstrate important facilities and their impact on symbolic computing over distributed computational infrastructures. Latest trends and technologies in the world of distributed computing emphasize the need for generic platforms, interconnection mechanism and standards that are only partially available in symbolic computations world. Due to these shortcomings, the aim of the EU Framework VI SCIEnce project (www.symbolic-computations.org) is to improve integration between CAS developers and application experts. The project includes developers from four major CASs: GAP [3], Maple[10], MuPAD[13] and Kant[21]; plus application experts organised through the international Research Institute for Symbolic Computation, RISC-LINZ. Its main objectives are to:

1. Develop versions of the CASs that can intercommunicate via a common standard Web services interface, based on domain-specific results produced by the OpenMath [184] and MONET projects as well as generic standards such as WSRF;

2. Develop common standards/middleware to allow the production of Grid-enabled symbolic computation systems;

3. Promote and ensure uptake of recent developments in programming languages, including automatic memory management, into symbolic computation systems.

The work presented in this thesis is mainly concerned with achieving the second objective, that of providing Grid-enabled symbolic computations in the context of novel framework SymGrid [110].

The main goals of the SymGrid related activities are:

1. Produce a portable framework that will both allow symbolic computations to access Grid services, and allow symbolic components to be exploited as part of larger Grid service applications on a computational Grid;

2. Develop resource brokers that will support the irregular workload and computation structures that are frequently found in symbolic computations;

3. Identify a series of applications that will demonstrate the capabilities and limitations of Grid computing for symbolic computations.

These objectives cannot be achieved without introducing new higher-level middleware systems. By providing a new domain-specific framework for symbolic Grid computations we aim to supply a sophisticated interactive computational steering interface integrating seamlessly into the interactive front-ends provided by each CAS, and providing simple, transparent and high-level access to Grid services. By defining common data and task interfaces, we provide the computational infrastructure to allow complex computations to be executed by orchestrating heterogeneous distributed components into a single symbolic application. Due to the generic interfaces build in the context of SymGrid we also anticipate that our framework can be further used for other application domains.

The SymGrid-Services component covers all the interfaces, discovery and composition mechanism and data models that are relevant at Grid level. The complementary

component within SymGrid that allows symbolic computations to be executed as high-performance parallel computations on a computational Grid, namely the SymGrid-Par component of the SymGrid framework is described elsewhere [200]. While there are several parallel Computer Algebra Systems suitable for either shared-memory or distributed memory parallel systems, work on Grid-based symbolic systems is still nascent. None of the systems implemented prior the ones provided by SCIEnce conforms to all three of our basic requirements:

- Deploy symbolic Grid services;
- Access available Grid services from within the symbolic computing system;
- Couple different Grid symbolic services into a coherent whole.

In addition to dealing with these key issues, a number of major topics are addressed by SymGrid architecture. Amongst the most important requirements are mechanisms for adapting to dynamic changes in either computations or systems. This is especially important for symbolic computations, which may be highly irregular in terms of data structures and general computational demands, and which therefore present an interesting challenge to current and projected technologies for computational Grids in terms of their requirements for autonomic control.

1.2 Contributions

The objective of this thesis is to investigate the potential benefits of distributed architectures for symbolic computations and to propose a novel framework that enables application specialists to exploit geographically dispersed computational resources. Communication latency, technologies used to interconnect remote computational resources, heterogeneity in hardware and software profiles are more likely to raise problems if geographically dispersed computational resources are used. As a result, specific solutions

tailored for such computational environments are provided. The work presented here lies at the border of two computational worlds as it combines the characteristics and functional requirements of both symbolic computation and distributed computing domains. The thesis makes the following contributions:

1. It analyses the general characteristics of research exploiting symbolic computation. Based on these findings it identifies the most important features of distributed architectures that could have a positive impact on the way research in symbolic computing area is conducted (Chapter 2).

2. We have designed and implemented a CAS Server as a collection of standard interfaces and implementations that make Computer Algebra Systems available for remote invocation and hence enabling their integration in large distributed architectures, such as computational Grids. CASs are the main software packages for symbolic computations. They are typically designed as command line interpreters and do not offer interfaces that enables them to be accessed remotely. The CAS Server defines autonomous computational elements that are able to expose the functionality of one or more CASs installed on the local machine or on the Local Area Network [58, 61, 148, 150, 129] (Chapter 3).

3. We have designed and implemented a novel framework for symbolic services orchestration, namely the Architecture for Grid Symbolic Services Orchestration (AGSSO). Complex symbolic computational problems may be usually decomposed and solved using a collaborative computational environment. AGSSO represents a viable solution for service discovery, orchestration and execution management of symbolic services exposed through CAS Server's interfaces [60, 61, 66, 148] (Chapter 4).

4. We have designed and implemented advanced mechanisms for controlling and managing the execution of scientific workflows. For scientific computations, the ability to control the execution of workflows by pausing, resuming, cancelling and

dynamically altering the execution path represents an important desideratum currently not implemented by any engine for Web Services orchestration. This thesis describes a custom solution that may be used to support the afore mentioned management capabilities and demonstrates its applicability in the context of symbolic services orchestration [64, 65] (Chapter 6).

5. We have designed and implemented a framework that allows Computer Algebra Systems to access generic Web and Grid Services. The interfaces exposed by CAS Servers allows seamless integration in Grid architectures and facilitates orchestration of computational resources. Its design makes it suitable for more advanced set-ups, while simpler solutions may be adopted to create and expose symbolic services. We present in this thesis a novel framework, Computer Algebra to Grid Services(CAGS), that enables CASs to access generic Web and Grid Services. This solution is especially useful for accessing any remote Web and Grid Service that do not comply with the interface proposed by the CAS Server [60] (Chapter 5).

6. We have designed and implemented an event based simulation framework that allows us to investigate the behaviour of the system in different environmental conditions. The process of testing and validating distributed architectures is difficult and error prone. The approach considered in this thesis is to develop simulated environments by replicating real life hardware infrastructures. This thesis presents a simulation algorithm derived from the event based simulation model and the results obtained through simulation [59](Chapter 6).

The main components of our architecture and the relation among them is presented in Figure 1.1. In Chapter 3 we present the CAS Server component which exposes CAS functionality through the interface of Grid Services. Chapter 4 introduces the AGSSO component and describes how multiple CAS Servers may be orchestrated to support solving of compound symbolic computation problems. In Chapter 5 we present the components that are required at client side to allow CASs to access functionality of

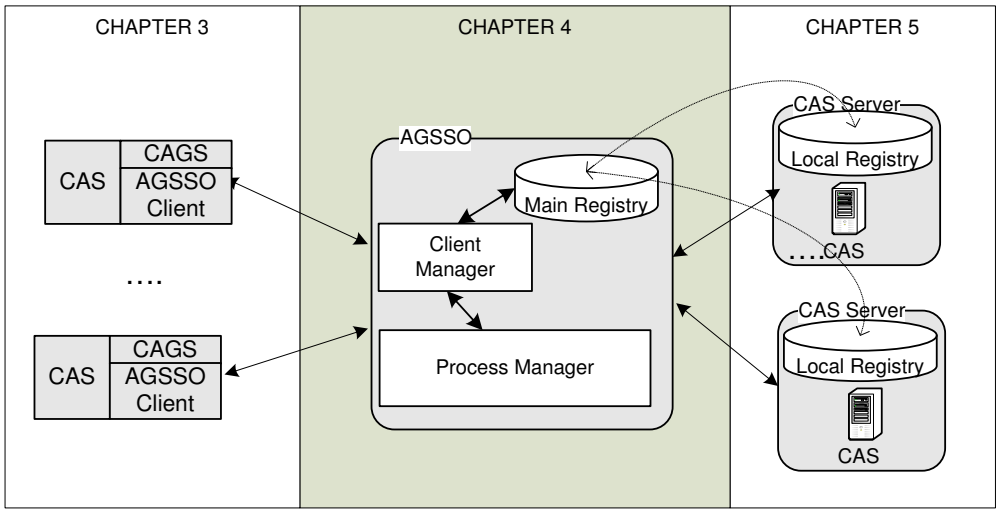


Figure 1.1: Main Components of the Architecture

remote Web and Grid Services. It also presents a solution for describing and submission of workflows for symbolic computations.

1.3 Authorship

Unless otherwise stated the work presented throughout this doctoral thesis was authored by myself and the work contained herein is my own. As a result of the research activities undertaken in the context of the European research project “Symbolic Computation Infrastructure for Europe” (SCIEnce) several research papers and technical reports were disseminated and software packages specific for the aim of the project were implemented. Part of these are directly related to the subject of this thesis. My contribution to the results presented within the publications I have co-authored is the following:

1. In [60] we have described a generic component that enables access to Grid and Web Services from within a CAS environment. The design that enables seamless integration with virtually any CAS was done by myself;

2. Within [58] we present a solution for exposing CASs through a generic Web Service interface that relies on a custom data model for encoding communication between a remote client and the server. The generic structure that allows multiple CASs to be exposed to a single interface and the custom data model to encapsulate communication are my contributions to this work;

3. Combining Web or Grid symbolic services exposed by the CAS Server was first described as a solution based on dynamic composition in [66]. As part of the results reported, the use of workflow patterns within the CAS to enable composition of symbolic services was my personal contribution. Additionally I have also contributed with the overall design of the AGSSO Server that allows dynamic composition of Web and Grid symbolic services. Previously we have reported a static solution for composing such services in [62]. Further, I have also designed the mechanisms that allow steering and management of computation for Grid Services that was reported in [64];

4. As a result of my work we were also able to develop a set of solutions for describing often used composition patterns in symbolic computations. These result were summarised in [65];

5. Data management across distributed environments represents an important topic. Based on OpenMath OMR reference objects we have designed and implemented a set of algorithms and components for seamless managements of data. The design of the algorithm and interfaces that address data management issues partially presented in [65] represent my personal contribution;

6. For testing purposes we have developed a simulation platform that integrates various components of the SymGrid-Services architecture [59]. The overall design of the platform and the way various components should be integrated within the simulation platform is also part of my personal contribution;

7. The contributions mentioned above were further refined and developed and the results were published in [61, 148, 98, 150].

1.4 Publications

The following articles were published during my research with the contributions from co-authors:

1. A. Carstea, M. Frincu, G. Macariu, D, Petcu, Validation of SymGrid-services Framework through Event-based Simulation, International Journal of Grid and Utility Computing, Vol. 2, No. 1, pp. 33-44, 2011

2. D. Petcu, G. Macariu, A. Cârstea, M. Frîncu:Service-Oriented Symbolic Computation, in Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications, N. Antonopoulos, G. Exarchakos, M. Li, A. Liotta, eds., IGI Global, 2010, ISBN 978-1-61520-686-5, pp 1053-1075.

3. D. Petcu, A. Cârstea, A. Craciun, A. Eckstein, Mathematics on the net: state of the art and challenges, Analele Universitatii de Vest din Timisoara, Vol. XLVII, Facs. 2, 2009, pp 95-116.

4. A. Cârstea, G. Macariu, M. Frîncu, D. Petcu, Description and Execution of Patterns for Symbolic Computations, The 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC2009), IEEE Computer Society Press, 2009, pp. 197-204

5. M. E. Frîncu, G. Macariu, A. Cârstea, Dynamic and Adaptive Workflow Execution Platform for Symbolic Computations, in Pollack Periodica, Vol 4, Number 1/April 2009 Akademiai Kiado, pp. 145-156.

6. A. Cârstea, G. Macariu, M.E. Frîncu, D. Petcu, Secure Orchestration of Symbolic Grid Services, Proceedings of High Performance Grid Middleware (HiperGRID 2008), November 2008, Politehnica Press - IEEE Romania Section,Bucharest, pp 25-33.

7. A. Cârstea, G. Macariu, M. Frîncu, D. Petcu, Workflow Management for Symbolic Grid Services, The 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Workshop on Grid Computing Applications Development (SYNASC2008), IEEE Computer Society Press, IEEE Computer Society Press, 2008, pp. 373-379.

8. G. Macariu, A. Cârstea, M. Frîncu, D. Petcu, Towards a Grid Oriented Architecture for Symbolic Computing, The 7th International Symposium on Parallel and Distributed Computing (ISPDC 2008), IEEE Computer Society Press, 2008, pp. 259-266.

9. D. Petcu, A. Cârstea, G. Macariu, M. Frîncu, Service-oriented Symbolic Computing with SymGrid, Scalable Computing: Practice and Experience (SCPE), Vol. 9, No. 2, pp. 111-125, June 2008.

10. A. Cârstea, G. Macariu, D.Petcu, A. Konovalov, Pattern Based Composition of Web Services for Symbolic Computations, International Conference on Computational Science (ICCS2008), pp. 126-135, LNCS, Vol. 5101, Springer-Verlag, 2008

11. A. Cârstea, G. Macariu, Towards a Grid Enabled Symbolic Computation Architecture, 3th International PhD Symposium in Engineering, Oct.2007, Pecs, 1st Prize for the best presentation at the computer science section. Published in Pollack Periodica, Volume 3, Number 2/August 2008, 2008, pp. 15-26.

12. A. Cârstea, G. Macariu, M. Frîncu, D. Petcu, Composing Web-based Mathematical Services, The 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Workshop on Grid Computing Applications Development (SYNASC2007), IEEE Computer Society Press, pp. 327 - 334, 2007

13. A. Cârstea, M. Frîncu, A. Konovalov, G. Macariu, D. Petcu, On Service-oriented Symbolic Computing, The 8th International Conference on Parallel Processing

and Applied Mathematics, Workshop on Large Scale Computations on Grid (LaSCoG2007), pp. 843-851, LNCS, Vol. 4967, Springer-Verlag, 2008

14. G.Macariu, M. Frîncu, A. Cârstea, D. Petcu, A. Eckstein, Redesigning Parallel Symbolic Computations Packages, The 16th International Conference on Parallel Architectures and Compilation Techniques (PACT2007), pp. 471-471, 2007

15. A. Cârstea, M. Frîncu, G. Macariu, D. Petcu, Generic Access to Web and Grid-based Symbolic Computing Services: the SymGrid-Services Framework, Procs. ISPDC 2007, July 5-8, 2007, IEEE Computer press, ISBN: 0-7695-2917-8, pp. 143-150

Chapter 2

The Impact of Distributed Architectures on Symbolic Computation

This chapter reviews related work, as follows. Section 2.1 introduces symbolic computation and related issues. Section 2.2 reviews fundamental architectural styles and introduces quality attributes that should be considered for evaluating architectures for symbolic computation. Section 2.3 discusses distributed computing environments while Section 2.4 focuses on Service Oriented Architectures with emphasis on Web Services. In Section 2.5 we provide an overview of Grid technologies and we discuss the main solutions for developing Grid infrastructures. Encoding standards for mathematical content are presented in Section 2.6 and a summary of the important topics covered within the chapter is given in Section 2.7.

2.1 The World of Symbolic Computing

Mathematics is a fundamental research domain with great impact in science and knowledge in general. Few research domains can advance without proper support from mathematics and mathematical software. Unfortunately the development of this research domain has suffered from a lack of computational resources. In contrast numerical computation has developed more rapidly because numerical algorithms require less computational resources than symbolic algorithms and therefore they were easier to develop and implement. When both numerical and symbolic algorithms can be used to solve a problem the latter should be used if precise results are required.

Computer algebra is a field of scientific computation that lies at the border of two worlds. It connects the world of mathematics and mathematical algorithms and the world of computer science and software engineering. The main software tools that are used currently for automatic manipulation of mathematical formulae are Computer Algebra Systems (CAS), either general purpose such as Maple [10] and Mathematica [24] or specialised to a certain domain of symbolic computation such as GAP [3]. Modern methods for scientific discovery motivate the need for such software packages.

The emergence of software tools enables new methods of conducting research in symbolic computations and changes the nature and the size of problems that can be addressed. Using software tools represents a significant step ahead even if such tools can be used on single processor computers. While a single computer may be enough to solve small problems, for large problems the computing power of a single computer does not suffice. Despite the efforts conducted in numerous research projects, a computational platform for solving large problems that allows easy access to computational resources and provides seamless ways to describe complicated symbolic tasks that would be solved using massively distributed computational environments has not yet been fully developed.

One of the problems in symbolic computations is the fast growing need for computa-

tional resources as the complexity of the problem rises. Even if the size of input data and the result are small, due to intermediate expression swell, the memory available on a single computer may not be enough. In his PhD thesis Watt [195] emphasises that execution of symbolic algorithms is irregular respective the size of the input given. Even a small modification in the input parameters can cause enormous discrepancies in the execution time and the amount of computational resources required. Algorithms for factorization [50] and algorithms for computing Gröbner Basis [113] are two examples where expression swell and irregular execution times occur.

Successful parallelizations of algorithms in domains such as parallel arithmetic in finite fields [165], modular integer multiplication [78] and exponentiation [136] were reported. A parallel implementation of the Karatsuba algorithm for multi-precision integer multiplication is described [114]. One of the symbolic computational fields that may benefit from distributing computations over multiple computers is polynomial arithmetic. Problems such as identification of similar terms and Gröbner Basis are suitable candidates for the parallel approach, to name only a few. A more detailed survey may be found in [153].

Lack of sufficient memory and long running tasks that require a significant time to complete were identified as the main issues in solving complex problems symbolic computation problems [131]. These issues are the main driving forces that led to development of algorithms suited for parallel and distributed computational environments. Additional advantages that distributed systems may provide such as using computing capabilities deployed on remote servers or computational platforms that make possible cooperation between researchers may also have an important impact on the way mathematics is conducted. CASs initially developed to run on local computers have already done important steps towards using parallel architectures, network based and distributed systems but they still lack interoperability with other systems and an uniform approach that would allow access to massively distributed computing architectures.

To take advantage of the capabilities offered by new computational infrastructures and

technologies the symbolic computations algorithms and techniques must be adapted. The benefits of parallel and distributed architectures for symbolic computation were observed several decades ago [49]. The report emphasizes the importance of symbolic computations for science and in particular for research domains such as high energy physics, celestial mechanics, chemistry, biology, etc... It also summarizes several problems and offers several guidelines that we believe are still applicable for the current state-of-art [49]:

- Better platforms that integrate both symbolic and numerical capabilities;
- More effective methods for solving important scientific and engineering problems;
- Increase availability of cheap, high-performance hardware platforms for symbolic computations systems;
- Symbolic computation software is typically large, sophisticated and error prone. General problems identified for other large software systems are also applicable to systems for symbolic computations;
- More modular, reusable and high-quality software needs to be developed.

Some of the CASs evolved over time trying to overcome aforementioned problems but even if they were subject to an evolution process most of them were not able to keep pace with latest technologies in modern hardware and software systems. Even if Grid technologies are massively used for conducting research in a multitude of scientific domains, currently there is little support for using Grids in symbolic computation. The lack of proper analysis and repeated evolution steps has on occasions led to software systems that are hard to maintain, tightly coupled and with little capabilities to interoperate with other similar systems. The lack of modularity and standards used during design makes the evolution process cumbersome and therefore these systems are difficult to integrate with modern parallel and distributed architectures.

Evolution was mostly triggered as a response to the immediate needs of the research teams that implemented them. One such example is REDUCE [15] which was first developed by Antony C. Hearn for solving problems in higher energy physics which currently provides a much wider set of features due to the effort of the international community that got involved. Some of the CASs were designed to solve problems in a particular area of research and much of the effort was spent to fulfil their main objectives. Relatively little attention has been paid to interoperating between CAS. Even if latest technologies were used at the time of the implementation most of these technologies became obsolete. One of the capabilities considered not critical in the early stages of implementation for most of the small CASs and with great importance for modern systems is the support for interconnectivity with external software components.

Most of the competitive advantages that special purpose CASs have over general purpose CASs rely on custom implementations of algorithms for a particular area of research that often use custom data models for representing data and mathematical formulae. Two popular special purpose CASs are GAP [3] and Kant [21]. General purpose CASs include Maple [10] and MuPad [13]. The number of CASs is quite large and a more in-depth analysis of these systems is provided in [107]. As a consequence, interconnectivity among CASs can only be achieved if conversion components that would translate data from/to their internal representation model to other models understood by the communicating party are provided. Such components, known as *phrasebook*s [184], provide mappings between different data encoding models.

Implementation of specific add-ons and components for particular CASs may be proven to be a difficult task. In order to implement OpenMath [184] phrasebooks that translate mathematical objects encoded using internal formats to the OpenMath format or vice-versa, low-level details regarding systems' implementation may be required. The lack of well documented formal descriptions of the internal architecture of the system [52] makes this process difficult or even not feasible. Architectural styles may be used to provide a high level description of components and the way they interact. Any additional insight allows easier and more reliable evolution process while keeping the resulted soft-

ware consistent with its intended purpose and initial assumptions.

One of the most simple ways to allow system evolution is to decouple the presentation layer of the application from the layer providing the actual functionality. Thus the two separated components can independently evolve to respond better to users' requirements. Such components may be even hosted on different computers, thin clients on users' machines and complex processing components on servers. This approach could take out the problem of installing complex software components on users machines' since the thin client is only composed of the graphical user interface components. It also may improve efficiency since components may be installed on dedicated machines with special hardware configuration that support the computational requirements in terms of resources.

Beyond interface-implementation decoupling even separating monolithic components into several sub-components based on their intended functionality may be beneficial. Independent components of a system can be deployed on separate machines that are interconnected by a communication network. Dolan et al. [79] uses this model to implement tools for partial differential equations that could be invoked remotely using TCP/IP socket calls. This deployment model eliminates the need to install complicated and hard to configure software packages on users' client machine.

Distributed architectures for symbolic computations do not always provide a computational gain. Algorithms that require significant communication among the components involved in the computations are not good candidates for distributed environments. Significant communication has a negative impact on the overall efficiency of the computations since communication latency dominates the computational costs. Modifying existing algorithms for symbolic computations in order to efficiently use parallel and distributed infrastructures is not easy to achieve. Unpredictable data dependencies and data access patterns represent the main obstacles in efficient parallelization.

The next step in the evolution of symbolic computations is to adapt existing algorithms to be used in large scale distributed environments. Symbolic computation systems should

be able to use the tremendous computation power that today systems can offer. They should also be able to make effective use of existing capabilities provided by the vast variety of existing CASs. Creating a computational infrastructure that is able to combine existing systems with latest technologies in distributed systems is not a trivial task. To fulfil this goal efforts and expertise of computer algebra developers, symbolic computing scientists and software engineers must be conjugated.

Several important research projects were conducted in recent years to investigate various aspects of symbolic computations that have a direct relevance for distributed symbolic computations. Intensive research was conducted in the framework of the 'Mathematics on the Net' (MONET) [32] research project. Its declared goal was to develop a proof of concept system and related semantic Web features for solving mathematical problems. Its main focus was to develop means to effectively describe mathematical services and problems and to create resource brokers that would match problems to solve onto existing services. Part of the ideas of MONET were shared with another research project, MathBroker [57] that had as a goal the development of an infrastructure of mathematical services on top of existing Web standards.

'Grid Enabled Numerical and Symbolic Services' (GENSS) [128] was also a project to follow the ideas formulated in MONET for discovering and matchmaking of mathematical services. In the framework of the project they have developed mathematical services and indexing portals that could be used to discover symbolic services. Through the 'Internet Accessible Mathematical Computation' (IAMC) [124] project an architecture to support distributed mathematical computations was proposed. Considering CASs as computational engine and Java technologies for developing network enabled wrappers, the JavaMath [170] API provides a recipe that would enable a developer to turn a CAS with no network communication capabilities into an engine capable to solve requests sent using RMI and XML-RPC technologies.

'Symbolic Computation Infrastructure in Europe'(SCIEnce) [19] is the latest research project aiming to develop a symbolic computational infrastructure based on the latest

developments in distributed computing technologies and particularly Grid computing. The aim of the SCIEnce [19] research project was to bring together the most important actors in the symbolic computational field and to find the most appropriate solutions to develop a viable computational infrastructure tailored to the needs of the symbolic computations field.

Apart from the research developments resulted from the projects mentioned above, initiatives to develop distributed environments for symbolic computing were also led by specific CAS system developer or 'ad-hoc' research teams. Important CAS vendors or even third party development teams have implemented specific tools and packages such as MathLink [193] and MathGridLink [178] for Mathematica [24], Grid Computing Toolbox [6] and Maple2G [153, 151] for Maple just to name a few. While their solutions may be applicable for specific cases they are not general enough to accommodate the variety of existing CASs, the fast changing distributed technologies. The capabilities they provide are limited with respect to support for describing complex workflows that should be run on distributed infrastructures, resource management and collaborative capabilities.

2.2 Architectural Styles and Quality Attributes

The successful development of complex systems cannot be achieved without a thorough investigation of the requirements that the system should meet and the available tools, technologies and implementation models that can be used to support those requirements. It is also important to have a good understanding of the advantages that particular architectural styles provide and for this reason we make a quick overview of the fundamental architectural styles that we use as foundations. During the design phase, quality attributes mentioned further in this section are used to motivate and support the decision to use a particular architectural style.

As defined by Garlan and Shaw [102], a software architecture represents a collection of

computational components that interact through connectors. A more recent definition states that *'The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them'* [46]. In software engineering, an architectural style *'defines a family of such systems in terms of a pattern of structural organization.*' [102]

The overall structure of a software system and its components has to be driven by efficiency characteristics that ensure that initial requirements are fulfilled. One of the first assessments on quality attributes and their role for software systems was done by McCall et al [132]. Of particular importance for research related software systems are *implementation attributes* and *runtime attributes* because they consider aspects closely related to the development and execution of such systems. *Business attributes* such as implementation cost and delivery time related attributes are of less concern and therefore they will not be considered in our analysis.

Some of the most important *implementation attributes* are [154]:

- Interoperability - the ability of a software component to be universally accessible to other components for the purpose of exchanging data. As we shall see later in this chapter, this ability is particularly important for establishing an infrastructure for symbolic computations;
- Maintainability and extensibility - the ease of altering the existing software implementation in order to correct or to extend the system's functionality;
- Reusability - the effort to adapt existing components so they can be re-used in more than one context;
- Testability - the ability to test and verify the correctness of the implementation;
- Portability - the ability of the implemented software to be deployed and used in conjuction with different hardware profiles and software infrastructures;

- Scalability - the ability of the system to handle increased number of requests. Usually scalability is attained by replicating subcomponents that interoperate for solving the incoming requests;
- Flexibility - measures the effort required to adapt the system for use case scenarios for which the system was not originally designed;

The most relevant run-time attributes of the software systems are:

- Efficiency - the volume of resources required by a software system to fulfil its function
- Availability - the property of the system to be up and running for long periods of time
- Security - the system property to enforce the required security requirements such as proper authentication and authorized access, the ability to handle malicious attacks, etc ...
- Performance - the ability of the system to respond effectively to high loads
- Usability - the degree to which the systems respond to users' expectations and to their level of expertise
- Reliability - the level of confidence based on the frequency of execution problems that arise during run-time
- Maintainability - the level of difficulty to which a running system may be modified, reconfigured or extended

Most large software packages combine pure architectural styles to achieve the desired functional characteristics. Different levels of abstraction of a software system may reveal different architectural styles. Most of modern architectural styles evolved from several fundamental architectural styles: *pipes and filters, data abstraction and object oriented*

organization, event-based implicit invocation, layered systems, data repositories, main program - subroutine styles to name only the most important ones. A more extended list of architectural styles and a more thorough description may be found in [102, 163]. In the following we present a short overview of the most important ones that inspired our design.

The *pipe and filter* is a software architectural style well known especially for its use in Unix systems. Computational components of the architecture, called filters, transform data received as input and feed the resulting data to the output. Internal state of filter components is not shared with other components and therefore the result of their processing is only based on the input values they receive and their internal implementation. To achieve interoperability filter components share the same model of data representation. In addition, reusability of components, maintainability at both design and run-time and the possibility to link components in parallel are also favoured by this design.

Sharing a common data representation may induce poor efficiency if significant processing has to be applied to data to transform it in order to accommodate the one used internally by the component. Because filters are tied to a common model for representing data, parsing from the common format to the one required internally and from the internal format to the common one may reduce the efficiency of the system.

The precursors of the *Data abstraction and Object Oriented Organization* architectural styles were *Main-Program-Subroutine* and *Remote Procedure Call* (RPC). The latter is an implementation variant of the former, tailored for network environments. The main idea that drives these styles is to partition the implementation based on the functionality they provide. As a result of this separation, maintainability and reusability of software components is improved. The RPC model was developed to allow parts of the software system to be executed on separate processors or machines.

Data Abstraction and Object Oriented Organization have several advantages in addition to the ones inherited from their predecessors. The components of the system are objects with a well described interface that encapsulate internal details, provide internal data

integrity and preserve invariants. The interface may be advertised separately from the implementation, allowing decoupling of the two in favour of maintainability and extensibility of the system. Interoperability is also easier to achieve because the client has a clear representation of object's interface. The main drawback of this approach is that the potential client must know in advance the identity of the objects it wants to access. Most of the systems implementing the *Data Abstraction* architectural style offer discovery mechanism that allow clients to find the object they require.

The *Event-Based with Implicit Invocation* architectural style defines two types of components: processing components and message managers. During execution, the processing components produce messages that notify interested third party components that a certain state was reached in the system. Interested processing components may register to be notified when events of a certain type of topic are reported. The role of the message managers is to receive produced messages and to notify interested subscribers that a certain event has occurred. These architectures offer the advantage of scalability and flexibility since new components versions and even new components can be easily added to the system.

The trade-off of the model described above is the lack of control over the order in which subroutines of the system are executed in the case of complicated execution scenarios. More than one subscriber may exist for the same event and the order in which they are notified is arbitrary. Therefore it is not possible to foresee the order in which processing steps are executed system wide after the occurrence of an event. Another important topic to consider is related to data exchange capabilities. The messages have the sole purpose of notifying that a certain event has occurred. Usually, in these circumstances additional data that describe the state of the system may be required, too large to be sent together with the notification messages. There are also situations when components waiting for the same events require different sets of data or representations of it. The solution is to add mechanisms through which components gather on their own the data they require. We also have to note that interested components must be alive and listening at the moment when the event occurs, otherwise they are not able to respond effectively

to the notified event.

Layered Systems represent an architectural style in which components are organised in layers. Each layer groups components that address problems of a certain type so that a certain layer offers functionality to the layer of components situated one level higher in the architecture and use functionality offered by the layer of components below. This type of organization favours separation of concerns for each layer of components, extensibility and maintainability because enhancements may be done at a certain level without affecting components at other levels and reusability of components. This model allows better communication between components and offers support for multiple message exchanges when a stronger coupling is needed between components.

The architectural styles described above represent only a small part of the ones that exist today. They are the foundation of modern architectural styles and offer important guidelines for further implementations. Architectural styles that are able to respond better to specialized architectures and functional requirements were created. A special category of such architectural styles is the one tailored for distributed computing.

2.3 Distributed Computing Environments

Distributed computing provides a viable solution for solving problems that do not fit in the memory of a single computer or are suitable to be executed using computational components distributed over a Wide Area Network, possibly in parallel. Collaborative environments even have a social dimension because they facilitate exchange of ideas and knowledge. Within this section we provide an overview of the main architectural styles and related technologies used for building distributed applications except Web Services which are discussed in Section 2.4 and Grids which are discussed in Section 2.5. The disadvantages that the architectural styles presented in this section have makes them less appropriate for building an infrastructure for symbolic computation.

According to Enslow [87], simple distribution of processing elements over a network must not necessarily be considered a distributed system. To fully leverage the advantages that distributed systems can provide, he emphasizes that a truly distributed system has to comply with the following rules:

- Processing elements to handle a task should be dynamically chosen; there should be more than one processing component capable to process a certain task and the system must be able to dynamically select the most appropriate;
- The computational elements must be autonomous and physically distributed over a network; autonomy guarantees a processing unit the freedom to admit or to refuse a request based on internal rationale;
- The system should have a high-level control framework that makes possible integration of distributed components into a whole;
- Services that are offered by the autonomous components should be identified using a naming scheme. A client must use the naming scheme to specify a service request while the control framework is responsible for mapping the request to the processing element. An important difference between network computing and distributed computing is that the latter uses machine names rather than IP addresses to specify the target machine that should handle a request;
- The components of the system should be able to collaborate to solve problems without a specific request coming from the control framework level;

The above definition is extremely restrictive and rules out a wide variety of distributed computing models. Autonomous behaviour is nevertheless of high importance if computational resources that are integrated in the distributed system are governed by separate organizations. Independent resource providers may still want to be able to control the way their computational resources are used. A definition that covers in a more loose way the notion of distributed systems is given by Tanenbaum [175]: *"A distributed systems is*

a collection of independent computers that appears to the users of the system as a single computer". As the author notes, two aspects are of great importance when characterising a distributed system. The first one is that the system is composed of autonomous entities. The second aspect that the compound nature of the system has to be transparent for the users of the system.

The architectural styles described in the previous subsection isolate software architectural characteristics from structural point of view. An important aspect that has to be considered in the context of distributed computing is the communication model used between computation components of the architecture. Variations of old architectural styles and new styles were created to respond to the new architectural constraints and requirements.

The architectural styles developed for distributed environments use as foundation the *client-server* style. The plain client-server style is based on request-response interactions that occur between the client and the server. The interface of the server components represent single points of entry which makes the server components easier to control and more secure. Software applications needed at the client level may be less complex and therefore easier to install and maintain. In order to execute the client side application fewer resources are required since most of the computational effort is now externalized to the server component. Data is usually stored and manipulated in a centralized way at server level, which eliminates the need for replica management and allows easier management of concurrent access.

The simplicity of the client-server model comes with the price of poor scalability. The maximum number of clients that the server can handle at the same time may be rapidly reached resulting in high response time or even denial of service. Scalability of the system is also poor if the internal components are tightly coupled. Modern distributed systems rely on client-server model to connect their components but the separation of function within the architecture is no longer evident. Some systems may use complex topologies such as centralised, ring, hierarchical, decentralised which are based on the

pure client-server architectural style. Hierarchical or decentralised topologies make no clear separation between clients and servers. An example is the peer-to-peer model in which components may act both as clients and servers.

Main-program-subroutine architectural style is implemented in distributed environments using the client-server style. The resulting style, Remote Procedure Call (RPC), allows a main routine to call a subroutine that is hosted in a different address space, usually a processing element hosted on a different machine. This model is the precursor of several distributed architectural models, the most popular being Web Services. More details on pure implementations of RPC can be found in [176].

Due to its advantages, object oriented programming (OOP) is currently the most used programming model. Similar to the RPC model, an application may use objects that are not necessarily resident on a single machine. The actual invocation of the methods of a remote object is transparent to the user and the communication logic is provided by the distributed framework onto which the application was build. The most popular models that provide support for distributed objects are CORBA [183] and RMI [9].

Common Object Request Broker Architecture (CORBA) is an initiative supported by Object Management Group (OMG), a not-for-profit computer industry consortium. One of the main purposes of CORBA was to create the premisses for inter-operability between applications developed in different languages. CORBA uses the IDL language to describe the public interface of the objects which makes their interface platform independent. The Object Request Broker (ORB) has the responsibility to find the actual object that must be invoked, to activate it if necessary, to pass values of the incoming parameters and to return to the client the result of the computation. Several languages have built in support for CORBA, such as C, C++, Java, Smalltack and several vendors have implemented ORBs. Unfortunately, the main goal of interoperability was not fully achieved because of lack of interoperability between various ORB implementations.

The Remote Method Invocation (RMI) has many similarities with CORBA but it does not offer support for interoperability with components that are implemented in other

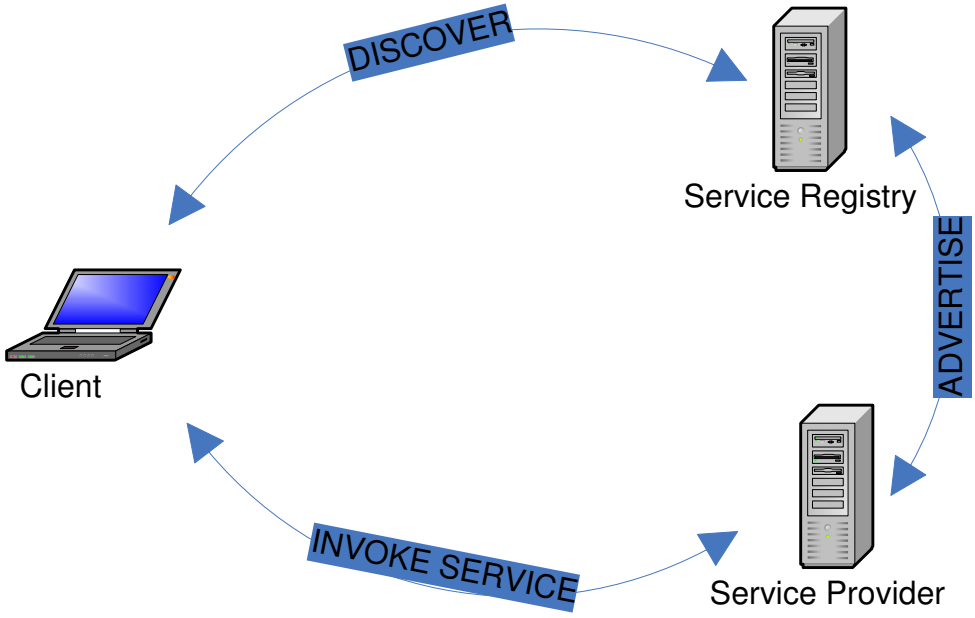


Figure 2.1: Service Advertising and Discovery

programming languages. Objects having predefined structure and advertised through the special service RMI Registry may be invoked remotely through proxies. Even if the actual implementation is specific, RMI uses the same pattern to enable remote invocation. An object design to be remotely invoked is registered with the RMI Registry service. A client queries the RMI Registry and obtains a local representation of the object implemented remotely. Any method invocation on the local representative is translated through the network to the remote object which executes the request and provides the result of the computation.

The general implementation pattern is therefore similar for both RMI and CORBA and it is also used in the case of Web and Grid Services. A client that wants to invoke a procedure/method implemented on a remote machine obtains a handle by querying an index service as described by Fig. 2.1. Unfortunately all implementations described above provide limited discovery mechanisms that enable a client to choose the best service to

invoke based on the service description. Web Services were developed to offer similar support while improving interoperability and discovery capabilities. We address Web Services and their advantages in more detail in the following section.

Multi-processor computers and clusters are especially designed as multi-threaded parallel architectures. Although their components are not geographically distributed they share most of the characteristics of a distributed system. Every processor may have its own memory space and computational elements may be connected through a Local Area Network (LAN). The small communication latency of such systems makes them suitable to solve problems that require intense inter-process data exchange. There are two base models of communication that may be used efficiently for LAN based distributed systems: distributed shared memory and message passing.

The distributed shared memory model provides an extended address space through which processors may access a shared memory pool. This pool is obtained by integrating the local memory of the participating computing elements. The extended address space is transparent for the user and the underlying system mediates all read/write operations. Message passing may be use in combination with a distributed shared memory environment or stand-alone systems that do not share local memory space. If message passing is used, inter-process communication is achieved through message exchange. Specific programming models for this type of distributed system are PVM [104] and MPI [133].

Both PVM and MPI are libraries that offer the fundamental tools that allow a heterogeneous collection of machines to be used as single distributed parallel processor. They offer standard APIs and implemented subroutines that facilitate inter-process communication. The main process of an application implemented using PVM or MPI, also called master, controls the initial set-up of the execution environment, uses explicit calls to determine parallel execution and controls all message exchange calls. The slave processes explicitly requested to be created by the master process in the initialization phase have the purpose to receive and solve computations received from the master. As opposed to the client-server architectural style where the client and server are autonomous, the life-

time and the behaviour of the slave processes is entirely controlled by the master. Due to these considerations MPI and PVM are less suited for building Internet scale distributed computational environments.

Maximization of computational resource usage, especially of processing time is an important area of research for computational systems developed using clusters and multi processor machines. Software tools such as Condor [74] and PBS [47] manage such resources using scheduling and load balancing techniques to ensure optimal utilization of computational resources. The users do not have to determine themselves the most appropriate machines where their tasks should be executed. They submit the tasks to the task manager which is responsible for planning tasks' execution on the most appropriate machines. Using resource managers improves resource utilization but it also ensures better response of the system to user needs and eliminates job starvation.

2.4 Service Oriented Architectures and Web Services

This section briefly introduces Web Services and related technologies. One of the most important characteristic of the Web Services world is the use of XML for encoding data, description of data types and services interfaces. The most important concepts related to XML are presented in Subsection 2.4.1. In Subsection 2.4.2 we provide an overview of the main advantages that Web Services provide. We also give a short introduction on fundamental components and concepts that a Web Service is composed of. Web Services and Grid Services play an important role in our architecture. CAS's capabilities are exposed through Web and Grid Services so they can be accessed by remote clients as described in Chapter 3. As shown in Chapter 4 once exposed as Web or Grid Services, automated tools for composing their provided functionality can be further used.

2.4.1 Data Encoding Using XML languages

XML (Extensible Markup Language) is a model for describing data in a structured way. Documents containing data structured using XML can be easily parsed by automated document processors while its content is still human readable and therefore it is not solely intended for automated processing. Its simple structure and the set of rules which govern valid XML documents make XML languages suitable for machine-to-machine communication.

Due to its general acceptance as a viable solution for describing structured data a large number of technologies and software tools were developed. The most important related technologies related to XML are:

DTD is a set of declaration that describe the accepted structure of a XML document. The DTD imposes a precise description of valid documents in terms of nodes, attributes, references and their valid position in the document. Based on a DTD it can be easily determined if a certain document has a desired structure or not;

XML Schema also referred to as XSD is a newer and enhanced validation XML language for XML documents. XSD defines a set of basic data types, mechanisms to define complex data types based on the basic data type and even features that allow strict control over length and multiplicity of components of the XML document;

CSS and XSL-FO are two XML related technologies that allow simple rendering of XML documents to formats that are suitable for visual presentation. These technologies are especially useful for presenting data in Web browsers;

XQuery is an XML language that was created to enable users to extract from XML documents the data that meets certain criteria. This language is versatile enough to locate required information based on the position of the node containing the information and filters that may be applied to the retrieved data;

XSLT is a powerful language that may be used by specialised tools to transform XML documents having a certain structure into XML documents with a different structure based on the structure and information contained in the source document;

The XML related technologies are well established and stable standards that offer an important support for managing XML documents. Automatic software tools implementing these technologies offer efficient means to handle XML documents. The software tools implement functionality to handle standard features of XML but they also implement best practices for encoding data using XML. Best practices ensure that tools related to XML processing can be used.

The most common such tools for handling XML documents are specialized XML parsers. Parsers are able to determine if a XML document is well formed, to validate documents against DTDs and XSD documents and to construct in memory representations of the data contained in the XML documents. The two main parsing techniques for XML documents are SAX and DOM. Advantages and disadvantages of the two parsing models are generally related to the size, the structure and the purpose for which they are used. SAX parsers are recommended for large documents or for documents in which XML elements are nested on a high number of levels because they can minimize the amount of data kept in memory and therefore are are more memory efficient.

In-memory structures created by DOM parsers are well suited when all information contained by the document must be manipulated at the same time. Parsing XML documents is a costly operation in general and choosing the incorrect parsing technique can impact even more in a negative way the performance of an application. Generally the benefits provided by using XML in computer-to-computer interaction outweigh the additional computational cost of boxing/unboxing data.

2.4.2 The World of Web Services

The notion of *software service* refers to a set of software functionalities that are available for clients if they adhere to imposed constraints and access policies of the provided service. The main design goals of Service Oriented Architectures (SOA) is reusability. Autonomous software agents implement functionality which is provided to clients as services through well defined interfaces. Service's interfaces provide syntactic information regarding how correct calls should be formulated. Details that describe the semantic meaning of the expected arguments or obtained results are not part of services' definition. Details that describe the quality of service that the services should provide are also not part of the standard.

Building applications by composing existing services offers numerous advantages. The resulting applications are easier to maintain and test while new functionality can be easily added. The application becomes scalable since it is possible to create more than one service providing the same functionality. A key requirement for the success of the model is interoperability. The communication mechanisms, interfaces and data encoding models must be consistent for all services so they can be effectively reused.

The advantages offered by XML and related technologies recommend XML as the solution for computer-to-computer message communication and therefore it represents the foundation of the most popular SOA standard. The SOAP protocol represents the foundation of Web services. The architectural style that lies behind SOAP is the client-server style. Autonomous services may be invoked using the synchronous communication pattern. SOAP is similar to RPC because Web services provide functionality implemented as operations, in the same way that functions achieve it in conventional programming languages.

SOAP defines a communication standard for computation components that are able to interact using a Wide Area Network (WAN) and therefore is well suited to be used over the Internet. XML based languages specific to SOAP are used to define a message

container, referred as a SOAP envelope, that encapsulates the actual data exchanged between clients and the Web Services. The URL naming scheme is used to identify actual services but the interface specification is decoupled from the actual identity of the service that provides the implementation. A client can determine which service to invoke dynamically at the moment when the actual invocation is needed.

SOAP does not require that a certain transport protocol is used. The most common protocol used for exchange SOAP messages over Internet is the HTTP protocol but other protocols can also be used. For example, implementations that use the SMTP protocol also exist but they are less used because HTTP is far more popular for exchanging data over Internet. Most SOAP implementations available from various vendors are HTTP based. Amongst the advantages of HTTP is that the associated port 80 is one of the few ports that are open for communications even when the most restrictive security policies are enforced by communication firewalls.

SOAP envelopes are composed of two sections. The header of the SOAP message is an optional part of the envelope. It may contain several header blocks in which meta information regarding the envelope and further processing instructions may be put. The message body, which is mandatory, may also contain several body blocks. They contain relevant data that must be sent to the Web Service.

As mentioned above, Web Services are very similar to the RPC model in the sense that the message the client sends to the Web Service contains detailed information about a certain function/method that should be executed and the list of parameters that must be supplied to the invoked function/method. The interface of the Web Service is decoupled from the implementation and therefore the actual implementation of the service can be done using a large variety of programming languages.

The most common implementations use languages such as C, C++ or Java. The sole requirement is that correct formulated SOAP envelopes are sent to the corresponding communication port using the communication protocol supported by the service. The SOAP message must be a well formed XML document and therefore any data that must

be transmitted enclosed in the body of the message must be encoded accordingly. Special characters that have a specific signification for the XML syntax must be replaced with their respective encodings.

As the definition of SOA suggests, services must be explicit in declaring the interface of the service, at least at syntactic level. It must implement mechanisms that describe in detail the signature of the operations that may be called remotely. The SOAP protocol specifies that each operation is defined in terms of the input message it accepts from the client, the output message that the service returns back to the client as a result of the performed service and a list of possible faults that the service returns if errors occur during execution.

Description of Web Services interfaces is done using the WSDL standard language which is also a XML language. WSDL documents describe the syntactic structure of the Web Service's interface. Details regarding the signature of the operations are all described using the XSD elements. The construction mechanisms that XSD provides allow arbitrary complex data types to be described and therefore they do not restrain the generality of the interfaces. The XSD technology is not specific to any programming language, therefore using XSD favours interoperability.

Parameters having a complex data type, i.e. formed by combining basic data types, may be transmitted to the server encoded as valid XML if the interface requires so. Extremely complex data types may not be well suited for describing parameter types that an operation expects. Alternative solutions to this problem are either to use XML *ANY* element to allow any XML content to be passed through or to use plain string codification of characters. To apply the second solution the operation interface must be modified to accept a plain string as an input parameter instead of a complex data type. This approach is a deviation from the standard because it forces the client and the server to agree in advance about the structure of a message.

We exemplify below plain string encoding for a compound data type that represents a mathematical formula expressed as an OpenMath object. More details regarding Open-

Math encoding are given in Subsection 2.6.1. A typical representation in XML of the mathematical formula 1 + 1:

```
<OMOBJ version="2.0">
    <OMA>
        <OMS cd="arith1" name="plus"/>
        <OMI>1</OMI>
        <OMI>1</OMI>
    </OMA>
</OMOBJ>
```

If the desired parameter's structure for an input message is similar to the one depicted above, the WSDL document describing the service's interface must declare explicitly the structure of the complex type using XSD declarations. For simple data types the XSD declaration is straightforward while for more complex data types it can became cumbersome. An immediate advantage of using XSD is that supplied arguments can be checked at server side before any additional handling of parameters is done and an error can be immediately thrown to the client.

The flattened XML representation of the complex type preserves the original format but characters with special meaning in XML are replaced with their respective encodings. The OpenMath object described above is therefore transformed to:

```
&lt;OMOBJ version="2.0"&gt;
    &lt;OMA&gt;
        &lt;OMS cd="arith1" name="plus"/&gt;
        &lt;OMI&gt;1&lt;/OMI&gt;
        &lt;OMI&gt;1&lt;/OMI&gt;
    &lt;/OMA&gt;
&lt;/OMOBJ&gt;
```

where all '<' and '>' characters were replaced with there respective encoding "<" and ">". This approach has a negative impact on efficiency because the messages have to be encoded/decoded to/from XML format. Validation steps that are normally run at the Web Service's interface level have now to be applied explicitly by the application receiving the message.

Based on service's URL any client can obtain the WSDL document describing the interface. Based on the information supplied in the WSDL, the client can automatically create suitable messages for interacting with the service. Therefore any client capable of generating the correct SOAP messages is able to interoperate with the service. On the other hand, knowing the structure of the interface does not ensure that the provided arguments and the functionality of the service are the ones expected by the client. The WSDL does not provide information about the functionality and the QoS that a certain service implements. Therefore, the client may know how to formulate a call but it does not know the significance of input parameters and the meaning of the results it obtains. Furthermore, composing multiple services is not possible without additional semantic descriptions of the services.

2.5 Scientific Computing Using Grids

The term of *Grid computing* has emerged in the distributed computing world at the mid of 1990's. Its main goal is to create a distributed architecture in which clients use computation resources offered by providers in a way that is transparent for the client. The process of computational power acquisition should be provided by a intermediary software layer that is able to detect available computational resources and effectively combine them. At atomic level computational resources are abstracted as generic elements called resources. Each resource has a set of attributes and a set of functionalities they provide, which the Grid software layer must manage, trying to keep a perfect balance between producers and consumers [92]. This model is especially well suited for

managing collections of computational resources owned by different organizations that agree to share their resources for supporting a common goal.

Virtual Organizations (VO) [90] represent dynamic communities of producers and consumers of computational resources that share and use resources based on a predefined set of rules. The VOs are usually spread over multiple administrative domains and reunite computing power offered by computing infrastructure owners from simple personal computer to large campus domains and super computers. Participants willing to share computational resources are usually part of academic and research institutions. Companies and governmental institutions are more reticent about sharing their computational resources due to security concerns regarding sensitive data and the lack of cost models that can be easily enforced by current Grid technologies.

The main middleware products used currently to build Grid infrastructures are Globus [91] and gLite [5]. Their role is to provide a software layer on top of hardware components that implements a core set of management capabilities. As a result, the hardware component layer, also called the *fabric layer*, can be managed in a consistent way across the whole VO. The middleware also provides security mechanisms, job management features, support for data transfer through standard protocols and infrastructure monitoring capabilities. The application layer which sits on top of the Grid middleware layer, can immediately use the provided functionality without having to reimplement new ones. As a result, the Grid middleware represents the foundation for interoperability and security over the Grid.

Apart from implementation details and capabilities offered, one of the main differences between gLite and Globus is the set of technologies used for interconnection of Grid nodes. Services provided by gLite are implemented as daemon processes that listen on various TCP/IP ports. This means that for interoperability reasons, nodes of the Grid built using gLite have to share the same configuration pattern and make sure that the appropriate communication ports are open for communications. This is not always easy to achieve when resources are spread over multiple administrative domains with different

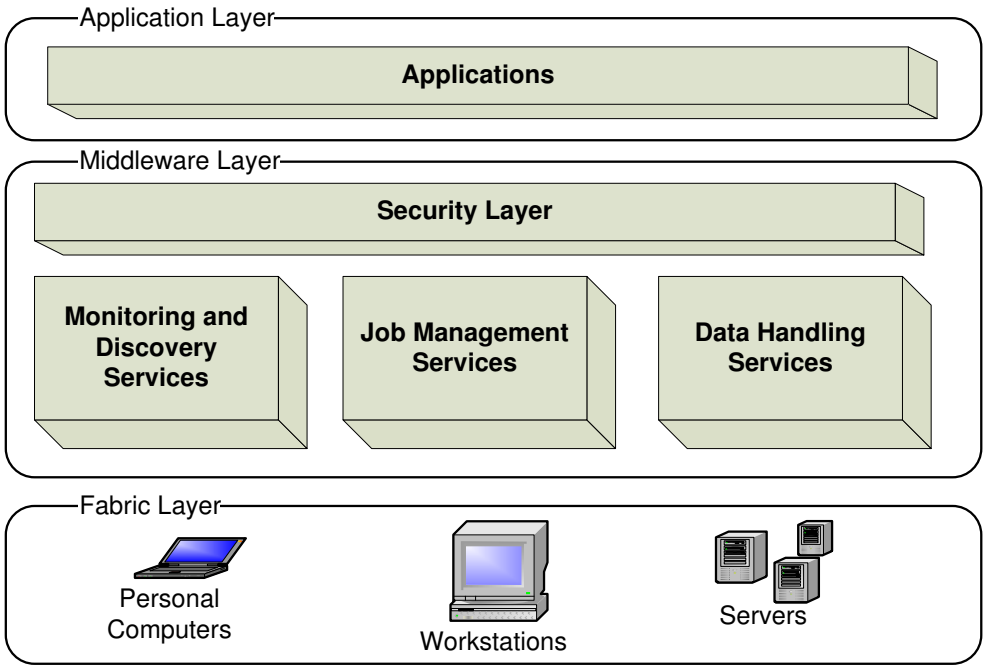


Figure 2.2: The Role of Grid Middleware

firewall configuration policies.

As opposed to gLite, Globus ToolKit 4 uses as communication interfaces modified Web Services and therefore all services may be available through a single interface advertised on the standard HTTP port 80. Management of provided services and their configuration are easier to achieve. Security risks are also easier to be managed because most of the services may be invoked through a single port which is by default open even in the most restrictive network firewall configurations. New services can be easily created and advertises in the same way standard Globus services are. Clients may easily discover services and syntactic information describing services' interfaces by retrieving their WSDL description document. There is also another important difference between Globus Toolkit and gLite. While gLite is oriented towards handling of tasks that are submitted by calling standard services, Globus allows service implementers to define new services and advertise them in a seamless way.

The model proposed by gLite is well suited for solving tasks, even compound ones, that rely on calling command line utilities installed on the computational node that handles the request. This strategy cannot be easily applied in the case of symbolic computations packages that are usually designed as interactive interpreters. Highly dynamic services configurations are also better dealt with by Globus which provides better discovery mechanisms. Due to its capabilities, Globus is currently considered the de-facto standard middleware for Grid computing and for the rest of this thesis the term *Grid service* will refer to the standard imposed by Globus Toolkit 4.

2.5.1 WSRF Compliant Grids

There is a clear distinction between Web services and Grid services and the role they play as distributed computing technologies. Stockinger [172] notes that both Web and Grid services were designed for wide area distributed computing. Typically, these services facilitate access to computation power and storage resources by advertising functionality using the same mechanisms. The most important differences between the two do not lie in the the way they are advertised, discovered and addressed but in their purpose.

The main purpose of a Web Service is to permit communication over the network between clients and service providers using standards that guarantee communication interoperability. The goals of Grid Services are beyond of those of Web services because they aim to offer mechanisms that allow interconnection of generically named Grid resources in one computational platform. For Grids, any computational resource, from processing units to printers and sensors may be abstracted based on their functionality and attributes. Grid services provide generic mechanisms to allow integration of Grid resources in wide distributed computing architectures.

The current standard for describing Grid services has its foundations in the Web Services Resource Framework (WSRF) [143]. The WSRF standard describes a set of mechanisms for easy integration and management of resources in distributed environments that are

built based on the Web services standard. The first initiative to augment Web services was proposed by the Open Grid Services Architecture (OGSA) [93] but the initiative was only an intermediary step towards the WSRF compliant Grid Services. With WSRF, Grid services have integrated the best features from both Web services and OGSA services worlds: on one hand the interoperability and on the other hand a mechanism to allow persistence of state at the service level. As noted in [172], a Grid Service is in fact an augmented Web Service that implements mechanisms for storing state information persistently beyond the lifetime of a single request rather than transiently.

The newer REST [88] standard for Web Services requires that a request must specify all the information needed by the server to handle the request therefore no stateful information should be kept at server level. The benefits of using this approach [88] do not apply for Grid architectures due to their different aim. For Grids, statefulness is an important feature that prevents unnecessary network communication and data sharing even between multiple clients. The WS-Resource standard, part of WSRF, specifies a core set of XML languages to be used for describing resources and their properties and defines a set of standard management protocols that should be used in conjunction with resources. Each Grid service has to describe resources that are made available to external users as XML documents and each resource has to be uniquely identifiable. To access a resource a client obtains the identifier of the resource from a factory service and in subsequent calls uses the identifier to specify the resource to which the call should be applied to.

The extensions that Grid Services define on top of Web Services go beyond the syntactic level because they enhance the capabilities of Web Services with consistent mechanisms that services' clients can rely upon across all Grid services. The WSRF standard is therefore a collection of specifications related to the management of WS-Resources that are guaranteed to provide the same functionality across all Grid service providers. These mechanisms not only add capabilities that could be useful for the Web Services world but they modify the architectural model of the applications that are based on Web Services.

The first important change is introduced by the *WS-Addressing* [188] and *WS-Resource*

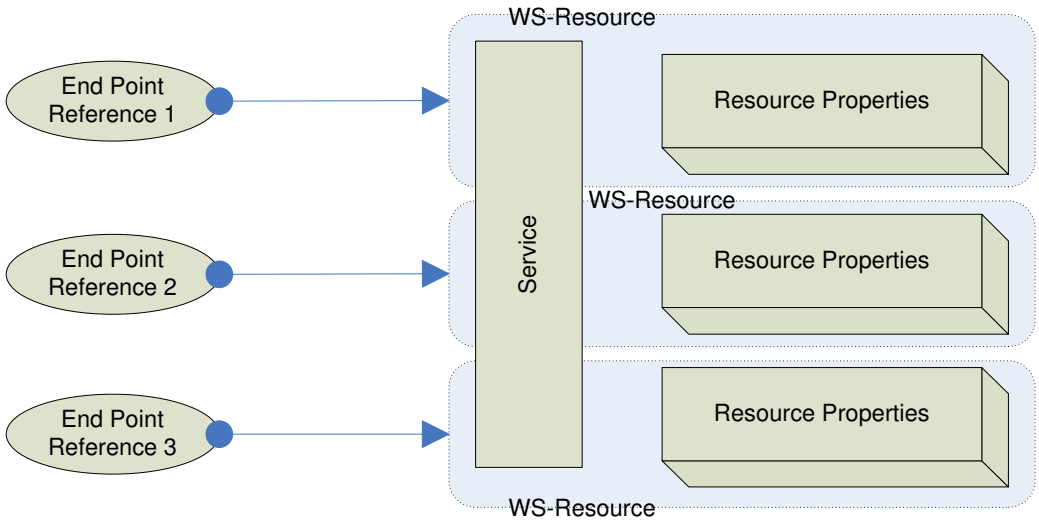


Figure 2.3: Structure of a Grid Service

[143] specifications. An instance of a regular Web Service is *stateless* and it is not designed to remember prior events. In order to create *stateful* services, state information which is stored at service level must be managed by the service and available for future invocations. Regular Web Services may be designed to implement such behaviour but the lack of a standard can only clutter the interface of the Web Service and complicate the invocation process because the client must send needed information explicitly.

The WS-Addressing specification defines a two level invocation mechanism that allows automatic attachment of a session identifier within the header of the message. Therefore the SOAP message is submitted to a URL that identifies the services and the header information is used at the service level to identify information regarding the session. Two invocations sent to the same service will differ in execution based on the state of the targeted resource.

Closely related to this mechanism is the specification that describes the structure of the persistent information that is stored as resources. Generically called resource, this concept can be used to describe any informational attributes of the entities that the service interface offers access to. While simple Web Services expose a set of operations that

an external user may invoke, Grid services are closer to the OOP paradigm. The attributes of an object are mapped to WS-ResourceProperties, and the whole structure of the resource is advertised within the WSDL document describing the Grid Service. The WS-ResourceProperties specification describes also the mechanisms that should be implements to allow seamless access to the content of the resource. Standard operations for setting and getting the properties of a resource may be part of the default interface of the Grid service.

A common implementation pattern for Grid Services is the *Factory Pattern* in which two services are used in tandem. The factory service is a stateless service that the client calls at the first invocation. The role of the factory service is to initialize a new resource object that is kept in memory and to send back to the client an End-Point Reference (EPR) that contains the necessary information to further interact with the new created resource. The EPR contains the URL of the service and an unique identifier of the resource.

The resource created and associated with the Grid Service is intended to outlast a single call. As a consequence, the Grid Service must implement life management features that control the lifetime of a WS-Resource and control in which circumstances the memory allocated for the resource must be freed up and the resource destroyed. The WS-ResourceLifetime [144] describes mechanisms that allow seamless management of resources lyfecicle which may be extremely complex. They are created and modified as a response to user's actions. Their lifetime spans over multiple user calls according to the purpose of the application. Unless kept alive by subsequent calls, the lifetime of a resource can expire based on the initial setting specified at the creation of the resource. For the resource to be destroyed after the expiration an explicit call to request the destruction is not required.

The interaction with a service should be standardized as much as possible to make sure that the aim of complete interoperability is achieved. Grid Services invocation may raise invocation exception that describe problems that prevented a successful execution. The WS-BaseFaults [141] specification provides a standard error types that may be used by

the application to inform clients that errors have occurred during the execution of the service.

The number of Grid Services that are exposed by a Grid node and the resources that are instantiated as a result of client calls may be high. Useful mechanisms that allow grouping multiple services together for easier management are specified by the WS-ServiceGroup [145] specification. Services can be added and deleted to a group and a group can be searched within a group based on a search condition. Besides describing the simple mechanisms to manage services the WS-ServiceGroup describes a set of guidelines on effective service grouping and management.

The Grid paradigm foresees the creation of complicated computational infrastructures based on the resources and their associated services participating to a Virtual Organization (VO). The aim of such infrastructures is to provide a suitable environment for solving large scale problems. The tasks to be executed to solve large scale problems may require a long time to complete. Therefore, asynchronous calls should be supported by Grid systems to support non blocking computation flows. The WSRF describes such mechanisms as part of the WS-Notification [142] specification.

WS-Notification defines two types of services: notification producers and notification consumers. Consumers register themselves with one or more producers to be notified in case a specified type of event occurs. Typically, the consumer registers to be notified for changes that occur in a certain WS-Resource. Any update in the internal state of the resource can therefore be advertised to interested consumers. This behaviour is especially useful with long running tasks, such as the ones that often occur in scientific problems.

One important aspect in distributed architectures is the need for a client to know the address of the remote service. The address can be known by default or the client can be expected to discover the services that provide required functionality. The standard discovery mechanism used by plain Web Services relies on UDDI [22] registries. The level of cohesion between the Grid Services is bigger than the one of Web Services due to the fact that they are part of a certain VO. Another significant difference is the

additional information that is stored in associated resources which can be meaningful for the discovery process. As a result, the mechanisms of discovery implemented by UDDI registries were replaced by a set of hierarchical *index services*. The services are compliant with the WSRF specification and standard enquiry calls can be formulated to retrieve information.

2.6 Encoding Standards for Mathematical Content

Various systems, tools and frameworks have been developed to ease the process of describing and solving mathematical problems. CASs are the most important ones but other tools such as QMath [14] and Sentindo [18] play an important role. Apart from their core functionality, an important requirement for such systems is to offer features or to support the process of exporting and importing mathematical content. Scenarios in which the user would want to export mathematical content for later reference or to enable results dissemination are part of the every day usage patterns.

Mathematical notation was developed over the years and it represents an important characteristic of mathematical formulae. Symbols and notation elements that are specific to mathematical writings cannot be easily replaced by function names without negatively impacting the legibility of the formulae. Unfortunately these special mathematical symbols cannot be stored in text files without converting them to a text format. The first solution to this problem was to replace symbols with string character function names but other solutions are currently considered more efficient and versatile. Due to rapid development of Word Wide Web related technologies, standards that use XML languages were preferred to plain character encodings. The main reasons are better parsing support that is readily available for XML documents and easier integration with Web technologies such as HTML pages that facilitates displaying mathematical formulae in Web Browsers.

While binary formats may also represent a solution for machine to machine communi-

cation, their disadvantage is that they are not human readable. Another disadvantage is that most of the binary formats are proprietary and only supported by specific systems or platforms. Mathematical formulae described in terms of OpenMath objects can be stored both in XML and binary format, both standardized.

2.6.1 OpenMath

One of the most used standards to describe mathematical formulae is OpenMath [184]. Its main aim is to provide the necessary mechanisms to describe mathematical formulae and encapsulate in its description semantic meaning of the mathematical terms. The semantic information ensures that a document containing mathematical formulae can be correctly evaluated and understood independent of the of the software package that produced it.

It is often the case that a mathematical document produced with one software package has to be loaded and evaluated by another package, either of the same type or different one. For mathematical formulae, semantic mechanisms must allow different software packages to determine the meaning of the content they parse. For instance, in the formula that expresses the area of a circle $A = PI * R^2$ the meaning of various terms of the formula should be self explanatory for a trained human eye. A software system though cannot assume the meaning of the particular terms. It is significant to have additional information to allow it to determine that "R" represents a variable while "PI" is a substitute for the well known mathematical constant π.

The description model that OpenMath proposes is not necessarily tied to a certain encoding format. The two encodings that OpenMath directly supports are a XML based language and a binary format. Either of the two may be chosen, depending on the scenario in which they are supposed to be used. The binary format is more compact and potentially more suitable for machine to machine communication when the communication channels use raw binary format while the XML encoding may be more suitable for Web service related technologies.

The fundamental concepts that OpenMath is based on are *Content Dictionaries (CDs)*, OpenMath *symbols* and the concept of an OpenMath *object*. Similar to the mechanism of dictionaries used in every day speech that states the semantic meaning of a word in the dictionary, OpenMath CDs are collections of OpenMath symbols that have a particular meaning in a specific context. A software package understands a mathematical formula that is expressed using OpenMath only if it implements corresponding *Phrasebook*s that allow the system to transform the formula in the encoding model that it uses internally. In this case, the software package *supports* or *implements* the corresponding CDs.

Mathematical formulae can be encoded as compound OpenMath objects by combining basic OpenMath constructor objects and OpenMath symbols defined in OpenMath CDs. It is a common practice to group OpenMath symbols that are related in CDs covering a particular mathematical area. Grouping multiple symbols in CDs is a convenient way to organize OpenMath symbols. The OpenMath symbol is a mechanism to identify certain concept in that particular area of mathematics and ensures that any interpreter will consider the same semantic meaning defined by the associated OpenMath symbol definition.

There are two main types of objects in OpenMath. The first category comprises of basic OpenMath objects:

- Integer - any element that is part of the mathematical set of integers
- IEEE - any floating point number expressed using double precision format
- Character String - any character string
- ByteArray - any sequence of bytes
- Symbol - any symbol element that is part of a CD
- Variable - represent a place holder; it has to have a unique name

For an OpenMath symbol to be correctly specified two mandatory attributes of the symbol have to be set. The *cd* attribute that specifies the OpenMath CD of which the symbol is part of; the *name* attribute is a meaningful name, unique in the context of the CD.

Compound OpenMath objects can be constructed by combining existing OpenMath objects. The constructive approach has to comply with the following composition mechanisms [184]:

- foreign(A) - is an OpenMath object if A is not an OpenMath object. This constructor function allows creating OpenMath objects from non OpenMath objects which may be useful if arbitrary data has to be encapsulated in an OpenMath compound object;
- application(A1, . . . ,An) - where A1,...An represent OpenMath objects specifies an application in a similar way with defining a regular mathematical function with multiple arguments. The first argument is referred to as the head. To encode a mathematical function the head object is an OpenMath object, such as an OpenMath symbol that specifies the function and the rest of the objects represent the argument that have to be applied;
- attribution(A,(S1,A1), . . . ,(Sn, An)) - where A,A1,...An represent OpenMath objects and S1,...Sn represents OpenMath symbols; this construction may be used to add attributes or characteristics that are part of the A object's definition;
- binding(B,v1, . . . ,vn,C) - where B and C represent OpenMath objects and v1,...vn represent OpenMath variables; it may be used to express functions or logical statements;
- error(S,A1, . . . ,An) - describes an OpenMath error objects

Based on the mechanisms described above computer algebra application specialists have created a strong foundation that can be used to describe complicated mathematical formulae covering the most common mathematical areas. Due to the popularity of XML

languages and the available support for parsing XML documents most of the OpenMath encodings are done in XML format. The OpenMath standard endorses the XML format and describes a set of language elements that correspond to the encoding models described above. To encode basic OpenMath object the following XML tags have to be used:

- Integer: <OMI>...</OMI>
- IEEE: <OMF>...</OMF>
- Character String: <OMSTR>...</OMSTR>
- ByteArray: <OMB>...</OMB>
- Symbol: <OMS cd="cd_name" name="oms_name"></OMS>
- Variable: <OMV name="variable_name"></OMV>

The corresponding XML tags that should be used to construct compound objects are:

- foreign <OMFOREIGN>...</OMFOREIGN>
- application: <OMA>...</OMA>
- attribution <OMATTR>...</OMATTR>
- binding: <OMBIND>... <OMBVAR>...</OMBVAR>...</OMBIND>
- error <OME>...</OME>

As an example, to encode in OpenMath the formula *sin(0)* where "sin" represent the sinus trigonometric function, the corresponding XML should be created:

```
<OMOBJ>
    <OMA>
```

```
        <OMS cd="transc1" name="sin">
        <OMI>0</OMI>
    </OMA>
</OMOBJ>
```

Remark. OpenMath objects are stored in separate XML documents. The corresponding XML document may contain a basic object or a compound object constructed using the mechanisms described above. For the document to be a well formed description of an object, its content has to be enclosed between the start and ending tags $< \texttt{OMOBJ} >, < /\texttt{OMOBJ} >$ respectively. These tags can only appear once in the same file.

OpenMath References

Due to their complexity, XML representations of large OpenMath objects can sometimes be large. It is also possible that some of the OpenMath sub-objects that an OpenMath objects is compound of may appear more than once in object's description. To shorten and simplify the representation of an OpenMath object, the OpenMath standard provides a reference mechanism that allows replacement of sub-object with a reference to the object's definition. Practically a reference replaces an in-line definition of the object and makes the encoding more compact and easy to read.

We illustrate this concept with an excerpt taken from the the OpenMath standard definition [184]. The following two encodings are semantically equivalent even if their definition is different. The first representation describes the mathematical formula "1 + 1" by combining in an OpenMath application object two OpenMath integers.

```
<OMOBJ version="2.0">
    <OMA>
        <OMS cd="arith1" name="plus"/>
```

```
        <OMI>1</OMI>
        <OMI>1</OMI>
    </OMA>
</OMOBJ>
```

The second encoding uses OpenMath references to replace the second definition the integer value "1" with a reference that points to an existing definition of the required object. Within the same document, the *id* attribute must have value distinct from all other identifier values. The unique value can be used to specify a reference encoded as the $<$ OMR $>$ XML element specified below.

```
<OMOBJ version="2.0">
    <OMA>
        <OMS cd="arith1" name="plus"/>
        <OMI id="bar">1</OMI>
        <OMR href="#bar"/>
    </OMA>
</OMOBJ>
```

The reference mechanism is similar to the anchor mechanism provided by HTML Web page description language. Valid references may point to objects described within the same document, objects that are described in a document stored at a location relative to the location of the file where the reference appear and even based on a absolute location. The OpenMath standard only requires that the value of the *href* attribute is a valid URI.

In the context of OpenMath XML encoded objects, reference resolving defines the process of identifying the OpenMath objects that are referenced by a compound OpenMath object's definition and, if the object is not hosted on the same machine, retrieval of referenced OpenMath object to make it available locally.

OMDoc

The foundation of OpenMath semantic annotation is the concept of content dictionary. Kohlhase [118] has investigated the suitability of using OpenMath for automatic proving. He concluded that the lack of semantics associated with OpenMath CDs makes the CDs machine readable but not machine understandable. OpenMath CDs were not conceived in a way that makes them suitable for computer to computer communication.

As a result, Kohlhase [119] has developed an extension of the OpenMath CD mechanism that allows clarification and addition of semantic context to CDs. The extensions define XML tags that can be used to accommodate several types of information such as semantic meaning of terms used in explanatory text elements and theory based classification of symbols defined in CDs and relation operators between theories. While these additions may improve mathematical formulae manipulation and automatic reasoning, these extensions were not widely adopted by computer algebra software packages.

2.6.2 MathML

MathML is a XML language that was created as a standard for describing mathematical formulae. Its main goal is to *"enable mathematics to be served, received, and processed on the World Wide Web, just as HTML has enabled this functionality for text"*[187]. The tremendous development of Web technologies and especially the intense use of Web pages to communicate ideas and knowledge motivated the need for standard languages to describe mathematical formulae so they can be understood and rendered by Web browsers. Since HTML is the most important language to define Web pages, MathML was built on the similar principles.

To encode mathematical formulae MathML provides two types of tags, presentations and content markup. The first can be used for encoding mathematical notation such as symbols while the second can be used for describing semantic meaning of mathematical

contents. These two types of markup are required because mapping between a mathematical notation and its semantic meaning is not always straightforward. A software system cannot automatically infer the meaning of a mathematical term only from its representation and therefore additional information has to be attached to the formulae description.

Even if a formula is described using semantic rich encodings, for rendering purposes, details on how to represent the data are still required. Even if two formulae are semantically equivalent, their term structure may be different. For such cases the rendering systems may choose one of the valid visual representations that its considered the most suitable one but this may not be the representation the user intends. MathML recognises these problems and includes in its standard capabilities for grouping semantically rich encodings with presentation content.

The set of markup elements that can be used for describing presentation covers the most important mathematical notations but it cannot be extended by a regular user. The rendering application has to recognize the markup and to render the formula based on the description. The use of a certain presentation markup element defines not only the position of a term in a formula but it also gives information about how the term should look like, e.g. the type of font to be used. Even if extension mechanisms would be provided for regular users, these would be too complicated to use and too difficult for a Web browser to follow them. The rendering model that is used by Web browsers is based on a set of conventions regarding how specific elements of the Web page have to be presented to the user based on their specific attributes and the context they appear.

For content description the MathML standard recognizes benefits introduced by OpenMath content dictionaries and provides similar mechanisms for semantic annotation. For conversion purposes, the standard even gives a comparison of the two and a mapping table that can be used for automatic conversion 2.1 between MathML and OpenMath elements [186].

As can be seen from the table above, the latest version of MathML provides good support

MathML	OpenMath
cn	OMI, OMF
csymbol	OMS
ci	OMV
cs	OMSTR
apply	OMA
bind	OMBIND
bvar	OMBVAR
share	OMR
semantics	OMATTR
annotation, annotation-xml	OMATP, OMFOREIGN
cerror	OME
cbytes	OMB

Table 2.1: MathML Mapping to OpenMath

for semantic annotation which was lacking in earlier versions. The base concept of content dictionaries can now be used in MathML similar to the way the semantic meaning is expressed in OpenMath. Due to these improvements it may be considered as a viable alternative to OpenMath. However, OpenMath remains the most popular data encoding standard for symbolic content and as a result it is the standard with the best support in the computer algebra computations world.

Existing computer algebra systems, especially the ones created for a special category of symbolic problems have custom data representation models. Their strength in solving the particular problems for which they were built comes from the data encoding and the special implementation of algorithms. Even if re-engineering was possible for this kind of system, it is not desirable. Therefore, interoperability of those systems may only be achieved by implementing translators from/to internal representation model.

2.7 Summary

Computer Algebra Systems (CASs) are the main tools for symbolic computations and of great importance for research world. Their first goal was to provide automatic tools

for handling mathematical formulae and to automate mathematical manipulations that otherwise would have been achieved with pen and paper. Lack of time and resources led to the omission of interoperability from their initial design. It is therefore difficult to use these systems in a collaborative way that permits solving symbolic problems that require computational resources not available on a single machine.

The need for better interoperability, better support for symbolic manipulation and standards describing mathematical content with semantic support were identified more than two decades ago [49]. Several joint research initiatives tried to offer viable answers to the problems that symbolic computations world has to face. Among most important research projects of the last years to investigate how distributed models can be used in context of symbolic computing are "Mathematics on the Net" (MONET) [11], MathBroker [2], "Grid Enabled Numerical and Symbolic Services" (GENSS) [7] and "Symbolic Computation Infrastructure in Europe"(SCIEnce) [19]. They have recognized the opportunities that distributed architectures may offer to symbolic computing and they have investigated solutions for creating distributed computational infrastructures for symbolic computing. The European project SCIEnce's aim was to provide the needed framework to bring together application specialists and researchers in mathematical fields. Finding the best solutions and technologies to create a symbolic computational infrastructure cannot be achieved without a coordinated effort of such interdisciplinary research teams.

A solid evolution of systems for symbolic computations towards a symbolic computational platform cannot be achieved without a thorough understanding of existing technologies and the benefits and shortcomings they introduce. Architectural styles used to implement software systems have a tremendous influence over their behaviour and limitations. One of the most evolved solution for building distributed infrastructures is Grid computation model. Its aim is to take advantage of the lessons learned from other technologies and to provide solid and standardized environments for building computational infrastructures.

The Grid model favours interoperability by providing a set of standards and software

utilities to cover fundamental problems that have to be dealt in distributed environments: security, resource and task management, data management. Due to the benefits it provides it may be considered suitable for collaborative environments as the one that symbolic computation requires.

Mathematical content exchange between software agents requires that the same encoding model is used by both communication parties. Moreover, syntactic level description of mathematical formulae is not sufficient for machine to machine communications. XML based languages are well suited for describing data that needs to be exchanged in distributed environments and several standard XML languages for mathematical content were developed over time. The OpenMath language is preferred for computer to computer interactions because it is a semantic rich language. For presentation purposes, especially for integrating mathematical content in Web pages, MathML is more suitable. An augmented version of OpenMath, namely OMDoc may be used for particular research domains such as automatic theorem proving.

Chapter 3

Exposing CAS Functionality as Web and Grid Services

This chapter introduces the CAS Server component [58, 61, 148, 150, 129]. Through its generic interface the functionality of multiple CASs can be exposed to remote clients. In Section 3.2 we discuss the requirements that drive the structure of CAS Server's interface. In Section 3.3 we describe the design of the CAS Server component and the way the CAS Server interacts with its clients and underlying CASs of which functionality it exposes. Available solutions for interacting with legacy software components and in particular with CASs are discussed in Section 3.4 while best technologies to be used for developing a distributed computation infrastructure are analysed in Section 3.5. The general structure of the request and response messages that CAS Server should handle are further described in Section 3.6.

3.1 Introduction

Most Computer Algebra Systems (CASs) lack capabilities that support building a symbolic computational infrastructure. The situation is particularly unfavourable for systems

that have started as small ad-hoc solutions and have evolved over time with improvements applied in a cyclic process. Even if the systems were improved over time, features that would allow these systems to interoperate with other software components were not high priority. Consequently, some CASs are only able to read input data from, and write output data to, text files stored in predefined locations.

Large symbolic computation problems may require computational infrastructures that provide large computational resources: high processing power, large memory and storage capabilities. Currently, most of the CASs are designed to be used as isolated software components and therefore they lack capabilities to access resources provided by massive distributed systems such as Grids. Some of the general requirements summarized in [49] such as the need for generally accepted standards for data encoding and support for interoperability capabilities are not yet fulfilled and represent major obstacles for building large scale symbolic computational infrastructures. The lack of support for modern technologies and standardization are main reasons for which we can include most of the current CASs in the category of legacy software systems.

CASs represent the main tools for symbolic computations and they cannot be easily replaced or reengineered due to the high level expertise required both in the general software engineering area and in the symbolic computational field. Therefore they still remain the main computational engines used for symbolic computations and solutions for large symbolic problems can only be build using these systems as foundations. To allow them to be part of large distributed architectures, CASs have to provide enhanced capabilities which can be added by applying modernization techniques. Adapting CASs to the latest technologies used currently for building distributed computational infrastructures is not an easy task.

The technologies used in software engineering have evolved tremendously in the last few decades and CASs have tried to adopt these changes in an evolutionary fashion. Technological advances and improvements in the way users interact with software applications may also be relevant for symbolic computations software. Watt [196] provides an

overview of various capabilities that could be easily fulfilled with a from-scratch implementation. Features such as visual manipulation of mathematical objects do not require fundamental changes to existing CASs. Support for a different data model than the one internally used may require fundamental changes which are closer to the core of the CAS.

Our main goal is to provide the blue print of a software architecture for symbolic computations and demonstrate its capabilities to support fundamental requirements of computer algebra specialists in terms of usability, efficiency, flexibility. On the one hand it has to provide infrastructure solving large symbolic computations, on the other hand it has to be versatile enough to permit easy adoption of new technologies and trends. Within this architecture CASs play the main role because they are the actual provider of symbolic computation capabilities. Additional components of the architecture will provide the support features that enables us to integrate CAS engines as a coherent whole.

Within our architecture the main components responsible for solving symbolic computations tasks are the *CAS Servers* which wrap and expose to remote clients symbolic capabilities natively implemented by CAS engines. The structure of the CAS Server's interface and the services it provides are primarily driven by symbolic computations related requirements. To a smaller extent, their structure is also influenced by the actual technology used to interconnect architecture's components. In addition to the core functionality of the CAS Server, complementary capabilities to ensure that CAS Servers are easy to integrate in collaborative environments must exist. Amongst them, indexing related services, security and task management are only a few of the functionalities that are required.

The general requirements and several fundamental features that the CAS Server offers are discussed throughout this chapter while more complex capabilities are described in the following chapters. In this chapter we investigate the most important requirements that have to be fulfilled to ensure adequate support for symbolic computing. Based on the main requirements we identify the most important computational elements that

an architecture for symbolic computations requires and the way these components are interconnected.

3.2 Top Level Requirements Driving the CAS Server Interface

Rich graphical interfaces able to display and manipulate mathematical formulae based on visual components are one of the features that makes CASs easier to use, more intuitive and even more attractive [196]. Providing symbolic support integrated with mobile devices may also be an interesting capability that would make symbolic computing more accessible to broader categories of users. Mobile devices such as mobile phones and PDAs were developed in the last decade to provide resources for software applications far more advanced than the ones required for device's basic functionality. They are now considered as viable tools for a wide range of applications and they can even be considered for solving small symbolic problems or as thin clients to server-provided functionality.

The most important challenge that the symbolic world still has to face is to provide support for solving large symbolic problems by enabling CASs to exploit and provide computational capabilities of massively distributed environments. Adoption of distributed technologies is currently the most affordable solution to build infrastructures that provide required computational resources. The most important issue that prevents seamless integration of CASs in distributed architectures is their lack of support for interoperability.

Immediate benefits of creating a distributed symbolic infrastructure are:

- Faster and potentially more accurate solutions can be obtained as a result of collaboration between specialized software packages and more general symbolic soft-

ware packages; particular sub-problems could be solved by the most suited software packages available even if the system that the problem was submitted to does not implement the required capabilities; through collaboration the capabilities of CAS can be easily extended by exploiting the capabilities offered by other types of systems, not only CAS [197];

- Knowledge bases providing already computed solutions to common problems obtained through long running computations could be reused by CASs and therefore the time required to compute could be in some cases reduced dramatically;

- Easier dissemination of results and collaboration between researchers by establishing shared environments and data repositories easily accessible from any computer with an Internet connection;

In the context of collaborative environments for symbolic computations CASs may play one or all of the following roles:

1. CAS as a client - CASs continue to represent the main environments used by researchers to formulate and solve symbolic problems. Depending on the nature of the problem and the computational capabilities offered by other CASs installed on the local machine or on remote servers, the CAS instance should be able to decompose the original problem into smaller parts and use external capabilities to solve them in the most effective way. External capabilities refer both to symbolic ones, provided by other CAS instances, or to other external capabilities;

2. CAS as a provider of computational capabilities; CASs capabilities could be used by external clients to solve problems of symbolic computational nature. These services could be accessed either by CASs or by clients that do not provide symbolic computation capabilities at all. The need of another CAS to request such services could be driven by its lack of a particular functionality or could be necessary for efficiency reasons. The provider may have better resources or it may just reduce the wall clock time needed for execution by executing subtasks in parallel.

3. CAS provider to CAS provider collaboration; based on the two roles specified above, we can easily imagine situations in which a CAS provider that handles a problem may have to collaborate with other CAS providers in order to solve a specific problem;

Interaction patterns supported by CASs provide important insights about how algebra specialists use such systems. The same patterns should also be supported by a distributed system for symbolic computing. The main interaction patterns supported by CASs were previously documented by Duscher [84]. One criterion Duscher uses to classify interaction patterns between the user and the CAS system is the number of messages that are exchanged during the execution of one task, not including the initialization steps. A second criterion used is the number of messages exchanged among collaborating CASs if such collaboration occurs. Multiple request-response messages exchanged within the same communication session require additional capabilities at both client and provider sides to ensure that messages are correctly interpreted in the given context.

A *Bilateral Simple Conversation Pattern* occurs if the client only sends one message containing the description of the problem to solve and the result is obtained as a response to the initial invocation. One of the systems that uses this conversation pattern is GAP [3]. In a more complex setup that permits collaboration among service providers *Multilateral Simple Conversation* and *Multilateral Simple Multi-Conversation* patterns may occur. The latter is a generalisation of the former and it is supported by systems such as Mathematica [24] and Maxima [12]. Within this patterns the server itself acts as a a client to other servers. To communicate with partner servers it may use one or multiple request-response messages.

The patterns described above are easier to handle because they do not require direct intervention of human users for manual steering of the computation. The *Bilateral Multi-Conversation* pattern captures the interaction model between the client and the provider for cases in which solving a problem requires additional data or steering during execution. Such conversation patterns can be for example observed in [12]. Depending on

the nature of the problem to solve, occasionally it may is possible to convert bilateral multi-conversation patterns to simple conversation patterns. This approach may be applied when it is possible to foresee situations where additional steering and data may be needed and modify the initial algorithm's implementation to prevent it. Because this approach is dependent on the nature of the algorithm and the CAS used, transforming Multi-Conversation patterns into Simple-Conversation patterns is generally unfeasible.

The computational elements of our architecture have to act both as service providers to clients and also as clients of other service providers either of symbolic nature or of another type. For solving a single task several components of the architecture may need to collaborate. Isolated interactions between the various components of the architecture are of client-server type but the architecture itself does not follow the simple client-server model since the communication pattern is more complex. One of the latest and most versatile architectural styles to support this type of pattern are service oriented architectures. Components act as independent service providers which may be combined to solve compound problems.

Existing systems for symbolic computations were buildt using a large variety of architectural styles and corresponding technologies. Their common goal is to provide an efficient way to access and combine capabilities of CASs to solve compound symbolic problems. To exemplify these architectural styles and the role their components have, we rely on following generic scenario. We consider that task T is composed of several subtasks $t(1),t(2)...t(n)$ which have to be executed in sequence. The output obtained from the task $t(i)$ represent data input for the following task $t(i+1)$.

Based on the capabilities that various systems for distributed symbolic computing provide, we identify two main architectural styles that were used to implement system for symbolic computations. In the first one the client component has an important role not only for describing the steps of the computations but also for selecting the appropriate services to invoke and for managing the execution process. In the first architectural style the client may be responsible entirely for finding services or specialized components

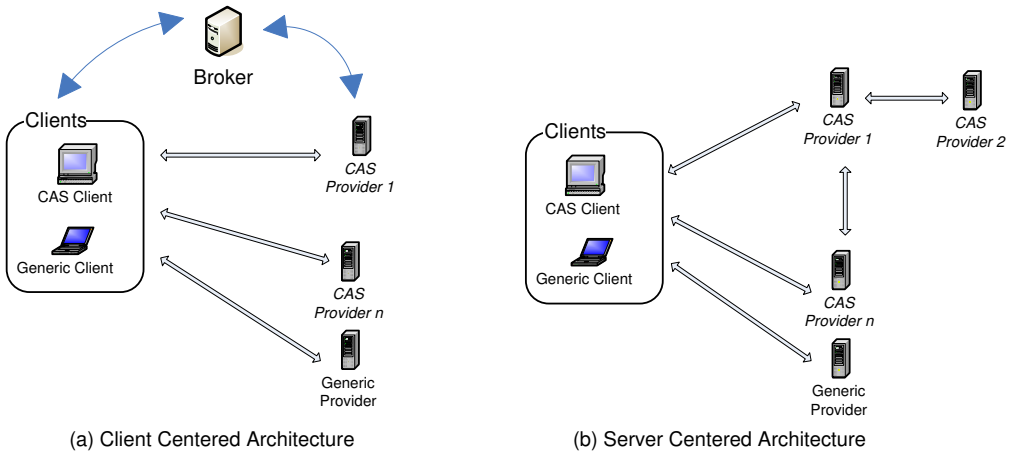


Figure 3.1: Server Centred Architecture

for indexing and selecting suitable services may be used. Due to the high importance that clients have during the actual execution this style is a client centred architecture. Opposed to this model, in the server centred architecture the client's role is to provide a description of the steps to execute. The whole management of the execution is done at server side by specialized components. The two architectural styles are depicted in Figure 3.1 (a) for the client centric style in which the client may use and intermediary broker and (b) for the server centric style respectively.

The most simple of the two is the client centric architecture which has its foundations in the client-server architectural style. In this architectural style client-server interactions are used. The client component has to identify by itself the service providers and to use internal rationale to determine the best provider to call if more than one such providers exist. It has to formulate request messages using the technology and message format expected by the server, to submit requests and to retrieve the results. These steps have to be done for each individual sub-task of the compound task and, if necessary, the client components have to combine results obtained using their own resources.

A slight improvement of the client centred architecture is obtained by introducing an additional component playing the role of a resource broker. Its role is to provide support for efficient discovery and load balancing of the executions in the system. The broker

analyses individual tasks and proposes a set of services that would be able to solve the request. The broker negotiates access on behalf of the client and provides credentials for the client to access the service. The client has still the responsibility to call the service providers, to collect result and to ensure executions steering based on obtained result. This architecture is slightly more efficient because during the selection process the broker may correlate requests received from multiple clients and provide an execution plan that improves resource utilization.

The second architectural style commonly used is more server side weighted. At the client side, the compound task has to be formulated in such a way that the user does not need to intervene during the whole computation process of the compound task *T*. Once the problem is properly described it is submitted to a server which in turn is responsible for managing the task, discovering the service providers that should be used and ensuring correct routing of the whole process. There are several important reasons, such as network usage efficiency or execution time further discussed below, for which the server centred approach is more efficient in terms of computational and communication efficiency.

The simplicity of the client centric solution has several shortcomings. The client has to explicitly invoke every server to provide the tasks and input data. Results that correspond to each subtask have to be managed by the client and submitted to the following server even if partial results are not of particular interest for the user. This negatively impacts the network load between the client and servers which is usually less reliable and slower than server to server communication links. For handling such results the client has also to provide sufficient hardware and software capabilities to handle and to process partial results that are not always available at the client side. Thin clients that could be run on mobile devices or on computers with small resources could not be used successfully if intermediate results are large, even if the initial problem and the final results could be handled.

An additional problem is generated by the nature of symbolic tasks which usually re-

quires long time to complete. If synchronous communication between the client and the server is used, the client must keep alive the connection with the server until the computation is finished and results are collected. Flexibility of the system is therefore affected. The solution to this problem is to use asynchronous communication and implement mechanisms for session management. Even if these are provided, the user has to periodically interrogate the server to find out the status of computations and if ready, to retrieve the results and continue with the following steps of the computations. In the absence of an available connection between the client and the server that allows the client to proceed with the next steps, the overall computation is delayed. Unnecessary idle time gaps occur between the tasks of the workflow which is not the case of server centric style in which the probability of the server to disconnect from network and to collapse is much smaller.

The server centric approach allows direct collaboration between symbolic computations providers. The task *T* is sent to the server which in turn manages the computation of the subtasks. Partial results do not have to be sent to the client and not even to the server that manages the computation if these results are not required for computation steering. They can be stored by the server that computed them and provided on request to other servers that require them for computing other tasks. Higher communication efficiency is thus obtained because the client only has to retrieve the final result. For both client to server and server to server communication asynchronous communication should be used.

Ideally, symbolic engines should be able to further identify subtasks of the initial subtasks *t(1)...t(n)* and automatically initiate calls to the most suited CASs to solve the particular subtasks. Unfortunately, existing CASs are not currently able to detect such situations and collaborate with other CASs. Automatic detection of subtasks of a task that could be better handled by another CAS requires fundamental changes that are not easy to achieve. A thorough investigation of CASs' capabilities for certain types of problems does not exist and therefore it is difficult to automatically identify the most appropriate CAS for handling a certain problem. Decisions regarding the best choice to

take are usually based on the experience and intuition of the computer algebra specialist.

One of the first successful systems to use CASs as computational engines was MathWebSB [94]. CASs wrapped as RMI and XML-RPC accessible services were integrated in a broader architecture with the aim to support automated theorem proving. Client systems of the architecture are able to discover appropriate services to invoke by interrogating predefined brokers. All brokers of the system are aware of each other and are able to exchange information about the services that are implemented in the systems. Therefore if a particular service is required and a broker is not aware of its existence it contacts other brokers until the service is located and a handle is returned to the client.

Part of the Esprit-OpenMath project, a client-server architecture that uses as main computational engines GAP instances demonstrates the viability of exposing CAS functionality while relying on OpenMath encoded messages [124]. These were the early steps towards enabling the GAP system to act as a symbolic computation server on one hand and to use other systems as clients on the other hand. Having the more generic aim to create a standard recipe to turn CASs in remote accessible computational engines, JavaMath [170] describes and implements a set of Java wrappers. GAP and Maple were used to create demonstrator wrappers. The resulting components can be accessed by remote clients through RMI calls.

Due to its popularity and its capability to provide a TCP/IP socket connection to connect to its core, Maple was used to demonstrate and build several remote accessible services. The system implemented by Schreiner [159, 160] uses multiple Maple instances and a distributed scheduler implemented in Java to which Maple instances submit job requests and from which the Maple instances may receive tasks to solve. The client Maple engine is responsible for defining the number of instances of Maple that should be used for the computation and which tasks should be executed.

A similar but more advanced set of software tools is the Maple Grid Computing Toolbox [6]. It may be used in a LAN of computational nodes on which Maple was previously installed to run Maple computations supervised by a master Maple instance. One of the

first initiatives to consider Grid technologies for exposing capabilities of Maple as Grid services was Maple2g [153, 151]. Through this package, Maple could be accessed as a Grid service and it was also able to access remote Grid services.

Another CAS system popular within computer algebra specialist community is the Mathematica system. Mathematica has seen the potential of distributing computations over a network and has provided the MathLink protocol for interconnecting Mathematica kernels more that one decade ago [193]. Another system, gridMathematica [8] is specially implemented for computing in cluster environments. As in the Maple2g component, MathGridLink [178] was implemented to permit access to Grid services implemented using early Grid middleware. It also allows the Mathematica user to deploy Grid services from Mathematica.

One of the most prolific projects of the last several years was the Monet (2003-2004) [180] project. The blueprint of the distributed architecture for symbolic computations they propose has as a central component the concept of service broker. Most of the research conducted under the Monet project was concentrated towards establishing a set of technologies to support intelligent service discovery and brokerage of mathematical services. The role of the broker on one hand is to store service descriptions of mathematical services, on the other hand clients interested in solving mathematical problems may contact those brokers to find the most suitable services for solving the problem.

The Monet project has heavily influenced and it was itself influenced by MathBroker (2001 - 2003, 2005 - 2007) [57] and GENSS (2004 - 2006) [137], two projects having similar aims to the Monet project. Among other results MathBroker proposed a model for describing mathematical services based on which the Mathematical Service Description Language (MSDL) [42] was developed. The GENSS project has used the ideas formulated in Monet and MathBroker to refine matchmaking techniques. The Monet project takes matchmaking of tasks to services a step further by describing solutions for the case in which a problem cannot be solved by only one service and a composition of services is required. The problem of decomposing mathematical problems into sub-problems that

can be solved by separate services was also investigated in the GESS project [137]. The foundations of the brokering and matchmaking techniques used within this projects use semantic Web technologies on one hand and specific mathematical techniques such as algebraic equivalence of mathematical terms on the other.

Although the broker introduced by Monet tries to provide a solution for the case in which multiple services have to be composed to solve a certain problem, the process of specifying such compositions is not easily accessible for regular users and composed services can only be deployed by service administrators. Even from the early stages of design of the SymGrid-Services component [110] the aim was to provide a computational platform to allow users to compose external services in a seamless way. Further development of the systems at the Grid level has focused on moving the responsibility of managing the composition of services from the client side to the server side where the whole process can be better managed. While the user still has the responsibility to specify the steps of the composition using high level constructs the actual composition is managed at the server side.

Beyond the high level constraints in the way the system is able to support symbolic computations that are inherited from the architectural style chosen, all of the above initiatives that tried to integrate multiple types of CASs to a common architecture had to consider the problem of interoperability. One important step that has to be made to achieve interoperability is the use of existing standards, and if such standards do not exist already, to provide solutions that are not biased towards a particular system. Generic solutions are more easily accepted and implemented by existing CASs. Exposing the functionality of CASs as services has to consider several important aspects: the structure of the interfaces accessible for remote clients, the data encoding model in which mathematical problems are formulated and service advertising and discovery mechanisms.

A viable solution for smooth transition to a distributed environment for symbolic computations is to adopt new capabilities in two evolution steps. The first one should concentrate on implementing software packages as external add-ons that would augment CASs

with capabilities to participate either as clients or service providers in distributed architectures. In the second step, benefits of various additions and features proven to be useful should be implemented as native capabilities of CASs where such evolution is possible. This strategy minimizes the delay in the evolution of computer algebra software packages and provides the opportunity to understand better how distributed computer algebra systems should support symbolic research field.

Not all features required have to be implemented by the CASs themselves. CASs should be responsible for the core symbolic capabilities while more generic ones have to be provides by the underlying software infrastructure. Integrating CASs with massive distributed environments requires implemented features that are not specific for symbolic computation and therefore, existing solutions for generic problems should be considered if they were already proven to work for other research domains. Similar to other computation domains, the raw computing power is provided by computers or specialized clusters belonging to research based institutions willing to cooperate for their mutual benefit. The resulted infrastructure is heterogeneous, highly dynamic and spread over multiple administrative domains. Fundamental capabilities such as communication, security or data transfer protocols are not in the scope of symbolic computing and therefore they should not be directly implemented by CASs.

Certain features required for interoperability are closely related to CASs and therefore they should be provided by CAS systems. For example, standard data models for encoding messages exchanged with partners is one of the features with an important impact on CASs capability to interoperate. Messages encoded by one CAS have to be properly decoded by recipient partners, irrespective of the particularities of the CAS or the machine they are installed on. They should be encoded using models that ensure that their content is properly understood and suitable to be exchanged in computer-to-computer communication. Even if internally CASs do not use generic encoding standards they should implement appropriate translators.

In most cases, CASs provide scripts or command line interpreters that allow users to

describe solutions of symbolic problems using the *routine-subroutine* architectural style. Standard functionality provided by CASs is usually available as packages of routines that users can combine or invoke. Mathematical formulae are encoded internally as mathematical objects and specific capabilities that allow their manipulation is provided by the system. Therefore, most of the algorithms and problems' solutions are described by computer algebra specialist as calls to existing functions or user defined functions that rely on core routines.

The most straightforward and convenient way to expose functionality of CASs as services is to provide means to enable remote users to access functions that are already implemented and available at the command line interpreter of the CASs. Therefore the RPC style is used to allow clients to executed functions implemented by a remote CAS. This solution favours usability because the same usage pattern applies for both remote and local invocations. A simple exposure of functions permits human users to formulate meaningful calls in the same way they would if the system was installed on the local machine. It is not expected that CASs should be able to cover the vast variety of technologies that can potentially be used for building distributed systems and as a consequence the solution is to provide a generic component that behave like a bridge between the CAS and the external world. To ensure interoperability with other systems, arguments of functions should be described using standard data encodings that are mapped by CAS onto internal encodings.

Due to the heterogeneous and dynamic nature of components and their internal configuration, indexing and discovery capabilities play an important role for efficient use of resources. Corresponding components that provide up to date information about available services ensure that the best resources are used in the most efficient way. Computational elements supporting the symbolic infrastructure should have an active role in informing preregistered service registries about their current state when significant changes occur. Even if such index services exist, their role is not to negotiate access on behalf of the client. Clients that already know which services to invoke should be able to do so without contacting index registries.

3.2.1 Requirements Summary

The CAS Server component must provide efficient and versatile mechanisms that allow remote clients to access functionality provided by existing CASs. Due to large variety of CASs, one of the most important requirements is to provide means to allow them to interoperate. As a result of our analysis we have determined several features that are needed to allow easy access to CASs provided capabilities and further, to allow automatic composition of their features:

1. CAS Server components must be autonomous and must allow access to one or multiple CASs through a single interface;
2. The interface of the CAS Server must be standardized for all CAS Servers of the architecture and not influenced by the CASs exposed through the interface. This requirement is particularly important to allow automatic composition of provided services;
3. The CAS Server must allow asynchronous retrieval of computed results and implement notification capabilities;
4. The CAS Server must implement mechanisms to support data exchange and collaboration;
5. Clients must be able to discover in a seamless way the list of CASs exposed by a certain CAS Server and the provided functionality;

3.3 CAS Server Design and Main Features

The previous sections present a high level overview of the main features that have to be supported by a distributed symbolic system and have drawn the guidelines that need to be followed. The CAS Server components act as mediators between the remote client and

the actual computational engines. Clients that require access to a certain functionality provided by the CAS engine do not need to call a different service for each particular function they access. The CAS Server provides a service that behaves as a single point of entry to which any request that describes a new task to be computed should be submitted. Internally, the request is routed to the appropriate CAS engine.

The relation between the CAS Server, client components, indexing services and as well to the various systems whose functionality it provides to clients is depicted in Figure 3.2. The role of the CAS Server is to expose CAS's functionality or of any other software package providing symbolic computation capabilities to remote clients. As shown in Figure 3.2, not only CASs may be exposed but other systems as well. For example, SymGrid-Par [200] is a framework that is able to manage installed CASs on a local cluster and even a Grid with the purpose of optimising execution time and usage of resources. As described further in this section, CAS Servers advertise their capabilities both through their interface or to centralised discovery repositories.

It is not uncommon to find personal computers on which multiple CASs are installed. As computational power becomes more accessible it is also a common approach to use multiple personal computers connected through a LAN to support a set of services and even to use more advanced set-ups such as computational clusters. For such hardware configurations it may be convenient to have more than one CAS installed on a particular machine or even have dedicated machines to host different CASs. Having a single point of entry makes possible to provide access to routines implemented by multiple CAS instances that are in the scope of the CAS Server at the same time. As a result the discovery and invocation process is easier for clients since there is only one service to invoke. At the service provider level, this approach provides the opportunity to use advanced solutions of job scheduling and load balancing.

The structure of the interface and the functionality that CAS Server provides is driven by the general requirements that we have discussed in the previous section: task submission and retrieval of results; discovery of the capabilities implemented by the service

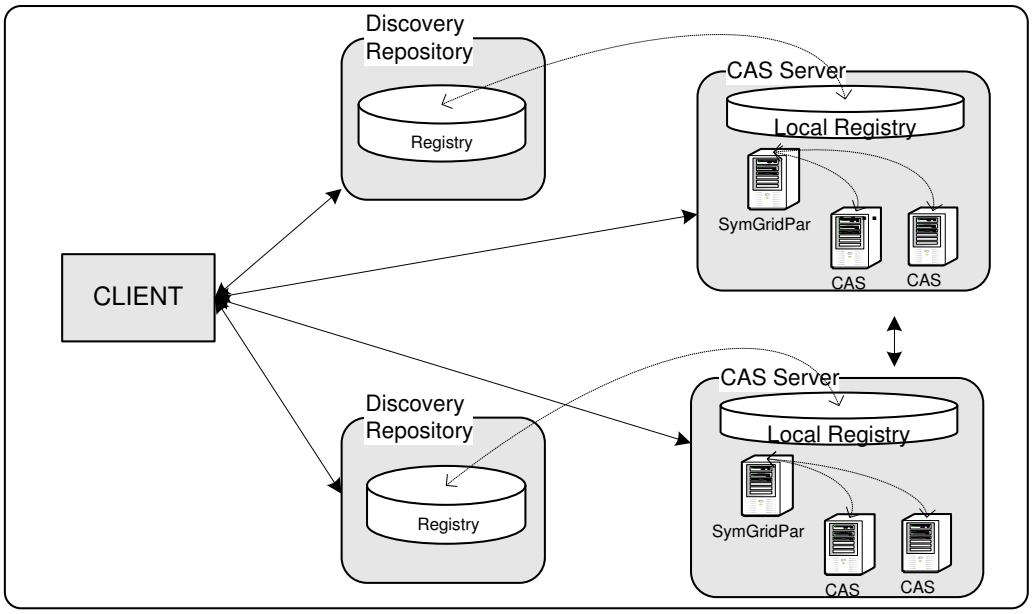


Figure 3.2: CAS Server and Relation to other Components

provides; support for task level management; support for data management and sharing. For all these features the common requirement is to ensure interoperability. One step towards this goal is to provide a standardized set of interfaces that do not vary over time which makes automated clients more easy to design and implement. Because the same interface is exposed by all CAS Server components, they can be easily interchanged or replaced.

Each of the requirements aforementioned is supported by a corresponding set of services. For submitting tasks and result retrieval the CAS Server is designed to support for two separated conversation patterns. The most common scenario is the one is which the client submits a task and receives an identifier that the client can later use to retrieve the result. This approach is suitable for submitting long running tasks because the client does not have to block while waiting for the result neither has to keep the connection alive. Task request can be easily sent through one device that is afterwards disconnected from the network. The result may be later retrieved using the same device or a different one. As an additional solution CAS Server can receive together with the task a URL address to which the result should be sent when the computation has finished. The capability is

particularly useful for dedicated servers that provide a service interface through which the result can be submitted. As we will see in the following chapter this features play an important role in automatic orchestration of services.

CAS provided functionality is grouped in packages of functions and exposing CAS's functionality through the interface is most conveniently achieved at function level. Virtually any routine provided by a CAS can be transformed into a service. Basically, a symbolic task submitted to a CAS Server has to provide as mandatory information the CAS type that the task should be managed by, the function name including its package name and the input values for parameters. Exposing each function as an entry on the interface would break the requirement for uniform interfaces on one hand and makes the discovery process difficult on the other hand. The tasks submitted through the generic entry point must adhere to one of the two encoding formats that we discuss further in this chapter, which are both OpenMath compliant.

Tasks that are submitted to a certain CAS Server are considered atomic in the sense that the CAS handling the task is not expected to further decompose the task and connect to other CAS Servers in the case it is not able to entirely solve the problem. Still, it may be possible that the task itself specifies that an external call should be made to another CAS Server, in which case an external call will be initiated. A current general limitation of existing CASs is they are not able to detect that a certain sub-task is better handled by another CAS and automatically forward it. Thus, we expect that for most of the cases, tasks are computed within the boundaries of the same CAS Server.

Using the CAS Server component to expose CASs functionality does not exclude the possibility of using specific systems for cluster and LAN level computation management such as the ones that SymGrid-Par [200] component implements. These systems have the goal of orchestrating CASs installed in a LAN to obtain efficient management of resources. From the CAS Server perspective they are seen as regular CASs and tasks received through the CAS Server interface are assigned to these components based on the same criteria used for selecting any other CAS. CASs themselves may be designed

to collaborate with other CAS instances installed on the LAN or to embed other CASs as internal engines. In such cases the way the CASs collaborate is not driven explicitly by the task's description. Features provided by software tools such as Maple Grid Toolbok [6] can still be used.

The manager sub-component of the CAS Server receives requests, decides whether the requested functionality is available locally and selects the most suitable CAS to solve the request. The selection process is based on information regarding the CASs and the functionality they provide stored internally by the *Local Registry* index component. From index the manager sub-component extracts information about which CASs are installed at the CAS Server, which is the physical machine that hosts it, which are the functions the can be invoked for a particular CAS and which physical resources are available.

The index keeps track of all information related to the hardware and software configuration of the CAS Server. As a consequence, the index also plays a role in the security of the whole system. Only functions that are registered to the index component by the CAS Server administrator can be called remotely. Functionality that should not be available for remote invocations for various reasons, including security reasons, are therefore not accessible.

Data required to compute a task is an important issue in symbolic computing. For large scale computations the amount of data produced and consumed by individual services that solve a large symbolic problem requires a careful consideration of data dependency problems. Tasks submitted to particular CAS Server may depend on data that is not stored within the CAS Server where the computation is done. Cooperation between the CAS Servers and intelligent handling of large data sets has to be supported by CAS Server components to ensure that at the time of execution all required data is available. A detailed description of data management scenarios that the CAS Server implements is provided in Chapter 6. For simplicity we exclude capabilities such as replica management and we rely on mechanisms to reference data sets that enable us to locate them in

a deterministic way.

Large symbolic tasks may take a long time to compute. There are often situations when such tasks have to be paused and resumed later or even cancelled. We group these related features under the generic term of task management. CAS Server supports task management features for the tasks that it executes based on features that are compatible with existing CASs. While operations such as cancelling a task are easy to implement by sending signals to corresponding CASs or by interrupting their execution, lack of support for check pointing at CAS level makes pausing and resuming tasks impossible in certain stages of the computation. Depending on the actual stage of computation in which a certain task is, the actions taken internally by the CAS Server may vary. Task management and related functionality is further addressed in Chapter 6.

Because the computational infrastructure we envisage is highly dynamic with regard to the actual CAS Servers available and the particular functionality that they provide, advertising and discovery mechanisms play an important role. Local Registry components depicted in Figure. 3.2 provide indexing capabilities in order to support the service discovery process. The CAS Server itself provides a set of services that may be used by a regular client to discover the functionality provided. The CAS Server is also able to notify dynamically interested third party components such as centralized indexes about any meaningful change that occurs within the server. Third party indexing components have to be expressly registered by the CAS Server administrator. To be able to receive update information they have to expose a predefined interface through which the CAS Server submits status updates.

Less related to the symbolic computations core requirements but playing an important role for system's interoperability is the middleware used for building the distributed infrastructure. Web Services are a good candidate for building interoperable components because they use standards that are platform and programming language independent. Due to their capabilities previously discussed in Section 2.4, they were considered as a viable solution for developing symbolic computational infrastructures by all recent sys-

tems including the ones developed within Monet [11] and MathBroker [2] frameworks. Built over these native capabilities, Grid services that are compliant with the WSFR standard provide support for interoperability, and additional capabilities such as security and data management. Important features that otherwise a CAS Server should provide by itself are already provided and ready to be used by the Globus Toolkit 4. Therefore the interface of the CAS Server is exposed as Grid Services implemented using Globus Toolkit 4.

Except from the GENSS project which has considered a single service to accommodate all the requests for symbolic services, the rest of existing solutions to expose CASs' functionality are using independent services. In Monet for instance, privileged users that want to create symbolic services have to create a set of files containing the code that should be run when the service is invoked and additionally, XML documents to describe the functionality and the interface of the service. Therefore, for each CAS that needs to provide support for a certain service an additional entry is declared on the interface [31]. One of the advantages of describing services using mathematical specific ontologies is that a formal description may be used to better evaluate the service and its capabilities but requires complicated matching techniques to discover which services to use. Within the system proposed by Monet, a client must properly formulate the problem it needs to solve and the broker must correctly identify the service. Matching is done using algebraic equivalence which may itself represent a symbolic computation problem sometimes impossible to solve [127].

3.3.1 Features Summary

The main features provided by the CAS Server component are:

1. multiple CASs may be exposed through a standard interface implemented using Grid Services technology;

2. The CAS Server acts like a mediator between remote clients and CASs installed locally. It provides capabilities that are not in the scope of CASs to allow their integration as computation engines in distributed computation environments;

3. The standard interface of the CAS Server provides support for: submission of tasks and retrieval of result; discovery of provided functionality through a set of indexing services; data management capabilities; task management features;

4. The CAS Server may use a variety of mechanisms to communicate with exposed CASs, e.g. TCP/IP socket connections and communication through pipes.

3.4 Solutions for Legacy Software Integration

Software architecture is determined by the components offering functionality within the system, connectors that link architecture's components and a control structure which imposes the behaviour of components within the architecture. As stated by Shaw et al. [163], two components that implement the same functionality might not be able to replace one another due to their particularities even within the same architecture. This may be due to particularities in their interfaces, the underlying hardware profiles they require, etc... Migrating a software system to a different architectural style demands that its components are adapted the new architectural constraints.

Legacy applications cannot be easily modified to take advantage of the new technologies. Software tools such as command line utilities that can be run on a local machine cannot be used in distributed environments without substantial reengineering. A potentially more effective approach is to provide adapter components that supply missing functionality that enables such tools to be connected to a distributed environment. Both reengineering and implementing adapter components for network interconnectivity has proven to represent major challenges.

One of the most important concepts driving software engineering is reusability as a solution for minimizing resources for implementing a software system. The process of integrating software components that were not built for interoperability is difficult even when the source code of the components is available [100]. Garlan et al. [101] have reconsidered the same problem two decades later and they concluded that the same problems make integration of components difficult even if technologies and standards have improved. For a complex system, it is hard to adhere to a single architectural style and in practice more than one is used to develop such a system [163]. Therefore, integration problems are not specific to situations in which legacy software needs to be integrated with new architectures and technologies. This type of difficulties arise especially when commercial off the shelf software packages must be used.

Software tools for symbolic computing may be considered legacy software because they generally lack the capability to interoperate with other similar systems and sometimes they even lack the capabilities to communicate in computer network environments. The main reason for this state of affairs is the way they were developed. Most of the CASs are the result of implementation efforts of small research groups aiming to solve particular classes of symbolic computation problems. As a result, some CASs evolved over time from scripts or command line interpreters to complicated problem solving environments. While the capabilities of these systems to interoperate are limited, the functionality they provide is by no means obsolete.

The high level of expertise required to reimplement some of the CAS software packages and encapsulated algorithms makes them difficult to reimplement or replace. Reimplementing CASs is also not an option because some of the features that would need to be adapted are exactly the ones that make some of the components be more efficient in solving a certain class of problems. An example is the data encoding model specific to a certain implementation of an algorithm which may have an significant impact on performance.

There are three important types of software evolution that generally occur in the lifetime

of software systems [73]:

- Maintenance refers to small interventions on the software system that are meant to improve the quality of a released software package by correcting minor execution bugs or by adding small pieces of functionality
- Modernization represent a more invasive approach with the aim to add more consistent enhancements but it may have a negative effect on the overall structure of the software system and its implementation
- Replacement involves total or partial replacement of the components of the system and for it to be successful it has to rely on a deep understanding of the original system and the functionality it provides

Code refactoring may lead to small scale modifications and therefore it may be considered maintenance or modernization or it can extensively alter the structure of a software system in which case a replacement of the old component was done. Adopting the most suitable evolution approach for a software system must involve a thorough evaluation of the value of the system in terms of usefulness, reliability, the level of coupling between components [155]. Based on this assessment replacement, reengineering or even keeping the system in its current state may be decided. The amount of effort already invested to develop CASs at the maturity level they are today and the fact that this software is mainly intended for research use makes replacement an unsuitable evolution strategy. Maintenance and modernization are more appropriate for software in the symbolic computations domain.

The immediate solution for integrating legacy software for which replacement and reengineering are not valid alternatives is to create wrapping components that act as adapters between the legacy components and the external world. The wrapper component fulfills the function that the façade software engineering pattern specifies because the wrapper has to achieve more that simple rerouting of calls. The wrapper technique has the advantage that the functionality implemented by the legacy component is still available

while the component can be integrated with other components using newer technologies without modifying the legacy component.

The level of insight available about components that need to be encapsulated is one of the factors determining the type of encapsulation used. It is also important to determine if atomic functionality implemented by the components must be exposed or it is sufficient to expose the functionality of the component as a whole. Several levels of encapsulation are commonly used: job level, transaction level, program level, module level and routine level [167]. The level of insight about a software system that is required for building a wrapper is lower for job level encapsulation and increases for the finer grained level encapsulations. The process, transaction level and program level techniques require less effort to implement and little to no intervention in the original source code and the way the wrapped components work. The software components are seen as black boxes offering predefined functionality. The data that is required as input and data obtained after processing are exchanged using the original mechanisms supported by the software. Usually these systems are able to communicate with the external world either through character streams or by reading and writing from/to files stored on the local file system. Migration of systems implemented using the procedural model require significant system analysis and reengineering. In order to avoid complete reengineering a wrapper based solution is also adopted in [75].

Based on the experience in modernization of legacy systems and integration of Commercial Off-The-Shelf (COTS) systems in custom architecture two important integration solutions are identified [73]. On one hand, *white box* modernization technique requires thorough knowledge of the system's internals. Good understanding of the general purpose of the system and supported use cases, its overall structure and its internals are required by this modernization technique. The encapsulation process uses the information mentioned above to decide exactly which components have to be encapsulated and if restructuring of such components is required or not.

The *black box* encapsulation technique has the advantage that it does not require thor-

ough understanding of the internals of the system and only about its behaviour supported through its interface. This approach considers the software component as a black box for which the only details available are the structure of the communication interface and the functionality the system is supposed to provide. Since neither documentation about the systems internal nor the source code are available, the integrator cannot make solid assumptions about the implementation quality and its behaviour. For complex systems even an intense testing of the black box system cannot possibly cover all the possible scenarios that may apply to the system.

Apart from the technique used to encapsulate the functionality of a legacy system there are two important elements that have to be considered. In order to be integrated with the rest of the components of the target system the encapsulated components must use a communication technology that is compatible with the technology used by the other components. With this requirement fulfilled, the messages that are exchanged between the components must be understood by both components.

It is often not possible to have the same internal data representation model for all the components of a software system. The data model used may be imposed for efficiency reasons by the internal algorithms. Therefore the only solution available is to implement a translator component that mediates the communication between the two components using different data encoding models. If the number of components that have to be interconnected is high a significant number of translator have to be implemented. A potential improvement considered in [70] for integrating legacy components is to use a single data model for the messages that are exchanged over communication channels and thus reduce the implementation complexity when having n components from $n*(n-1)$ to $2*n$.

Due to the importance of legacy software for both research and software industry the possibility to automate integration with new technologies was thoroughly investigated. The high variety of models and technologies used to implement software systems makes implementation of a universal encapsulation solution difficult to achieve. The encapsu-

lation process has still to consider several common issues:

1. The encapsulation technique refers to the direct means to communicate and leverage the functionality implemented by a legacy software component
2. The data model used by the encapsulating component, which is generally a more abstract and more versatile data model that the specific one used by the legacy component
3. The technology to be used for exposing the legacy functionality as required by the target system

The white box encapsulation provides better control over the behaviour of the resulting system but it requires reengineering effort while the black box technique is usually easier to achieve but less reliable. When access to the source code is provided the changes that ensure compatibility of the legacy systems to newer technologies can be incorporated directly in the legacy component and therefore we don't have to separate components, the legacy component and the wrapper, that are forced to use a potentially unreliable communication mechanism. If black box encapsulation is used, the wrapper has to accommodate the available communication mechanism that the legacy component provides and the underlying data model that the component is able to understand.

One of the most flexible communication mechanisms is based on TCP/IP. Its basic capabilities allows exchange of data formatted as byte streams but the preferred message encodings are based on XML languages. Available TCP/IP communication mechanism permits a high degree of freedom in choosing which technologies and programming languages may be used for wrapper implementation. Unfortunately there are many legacy systems that were designed to be used as command line tools and therefore the only mechanism that can be used for encapsulation is through communication pipes. It is a typical case of program level encapsulation in which the program is started as a process and its input, output and error streams are controlled by the wrapper. The process is fed

with the expected input values and the results are parsed from the values obtained from the output and error streams.

The legacy components are not only heterogeneous with regard to the communication mechanisms that can be used to interact with them; they are also heterogeneous with respect to the underlying data model that can be used to communicate with the legacy system. Most of the systems that can be used as command line tools expect as input a list of parameters that define the input values or system paths to files that store the necessary input data. Encapsulation of command line interpreters is even more difficult since they are interactive systems for which there is a close dependency between the input values supplied and output values obtained as a result of processing.

Developing generic methodologies and frameworks to encapsulate legacy components using the black box approach was considered by several research and industry projects. To overcome the inconvenience of multiple legacy components that have to be statically wrapped using individual wrappers, Fiesher et al [89] provides a script based framework that is capable to connect to legacy components for which adapted wrappers were developed and registered in advance. It uses black box encapsulation and a model of dynamic wrapper selection that allows the system to evaluate which legacy component must be called based on the external invocation parameters. Services offered by legacy components are exposed as CORBA methods and any call to such a method is mapped to a program level encapsulation that uses one of several possible communication modes: direct invocation, pipes, or socket connection.

A more fine-grained approach of encapsulation was considered in [41]. Using procedure level encapsulation legacy functionality written using COBOL programming language can be exposed and invoked through a Web browser. The system calls the individual functions implemented in COBOL with the corresponding arguments obtained from the client side through HTML forms. This model may be considered as a viable solution for exposing a small number of functions that are accessed independently and not as part of more complicated scenarios.

The development of Grid related technologies have motivated researchers to migrate their domain specific software in order to take advantage of the computation power provided by Grids. JACAW [111] is a tool that allows procedure level encapsulation of C/C++ based numerical and scientific routines. The encapsulation is achieved using the JNI technology provided by Java and the resulting components are adapted and can be used as standalone services or they can be composed using the Triana workflow manager. Adapting the legacy components, forces the use of the Triana data model for interchanging messages.

For command line utilities specific to microbiology research domain, SOAPLAB [161] provides a framework that allows easy integration of command line utilities in a Web Services based distributed computation environment. The wrapper modules implement simple program level encapsulation. Configuration files must describe for every legacy component details regarding the command line tool and the parameters that the tool expects when invoked. Among other features of the system, an API provides basic discovery functionality of the utilities that are registered to the system and the utilities are made available for remote invocations through Web service interfaces. Once the functionality is available through Web services other components can easily implement adapters to compose the functionality or to provide access using other distributed technologies such as Web pages and Web portals.

Nimrod [53] family of scientific software products is not only a package that is able to integrate command line utilities in Grid environments but also a platform that allows resource management and task scheduling over Grids. Wrapping mechanisms used by Nimrod are based on the program and job level encapsulation using a black box approach. One of its advantages is that it can be used in conjunction with a variety of Grid middleware products such as Globus Toolkit. Using Grid specific file transfer mechanisms the wrapper of a command line tool is able to transfer resources required for processing from other computational nodes.

A similar evolution path and set of functionalities apply also to the NetSolve/GridSolve

family of products [80, 162]. The initial intent was to use network capabilities to control and use in a collaborative mode hardware and software resources distributed over a network. The basis of the system was the client-server architectural model. As a proof of concept, the system demonstrated its capabilities to access remote functionality provided by a linear algebra package, LAPACK [38]. In the proposed architecture there are two important types of components. The first one is the server which exposes wrapped versions of locally installed routines. The second one is the agent which indexes existing services and assists the client in choosing the most suitable service according to its needs. Using task sequencing based on a Direct Acyclic Graph (DAG) representation of a workflow, the system is able to direct a list of interdependent tasks to a single server in order to prevent network traffic. The underlying mode is GridRPC which is based on function handles and session IDs.

Grid systems' most important aim is to share computational resources using mechanisms that make sharing resources transparent for the and user.An important support in this aim is offered by middleware packages for building Grids. Globus offers several important fundamental services that allow building up collaborative environments. Without extending Globus basic capabilities, legacy software suitable for program and batch level encapsulation may be accessed using GRAM managed jobs. A client may use the Resource Specification Language (RSL) to instruct Globus to execute a certain program or batch of programs for which a list of parameters should be fed. Apart from information that identifies the server machine, the executable to be started and its parameters, RSL job descriptor may contain meta information about the execution. Details such the number of times the executable must be run, the minimum and maximum amount of memory, the maximum time to run and other similar parameters may be specified.

The emergence of Grid technologies did not provide new models of integrating legacy components [125] and exising models still rely on black box wrapping. The mechanism offered by middleware components such as GRAM are task oriented and the user of the remote service has to provide details about the task and the application that is going to solve the task. The other model is based on exposing functionality of legacy systems

through a more friendly and easier to manage Web service interfaces. As a step further, basic management capabilities offered by WSRF framework such as the WS-Resource related technologies were considered in [125] as a base for more complicated problem solving environments.

Web services interface is also preferred in [106] as a viable option for creating a generic wrapper for legacy software. The generic wrapper still requires that the administrator builds control files to describe the interaction between the wrapper and the legacy application. The high number of initiatives trying to offer a solution for integrating legacy systems into distributed environments such as [168, 55, 36] indicate that a generic approach cannot be seen as a viable solution for all cases. Even if slightly generic approach may be considered, the high heterogeneity of legacy systems imposes tailored solutions for given situations.

Our solution uses the wrapping technique and exposes functionality of CASs through the interface of Grid Services. A custom interface and Grid Service is a more versatile and efficient solution than using WS-GRAM capabilities natively provided by Globus. To demonstrate this we have tested the access time required for a client to reach the CAS hidden behind the service. The test bad we used is a server PC HP ProLiant DL-385 with 2 x CPU AMD Opteron 2.4 GHz, dual core, 1 MB L2 cache per core, 4 GB DDRAM, 2 network cards 1 Gb/s.

The results obtained for the case when *RunCommand* was used to run GAP and Maple tasks shows an average of 51 milliseconds while for the WS-GRAM approach showed an average overhead of 678 milliseconds. The difference between the two approaches is significant when multiple invocations are needed, as in the case of combining several CAS functions. The invocation of the WS-GRAM service requires an extra Web Service call. It is then expected that using WS-GRAM induces some overhead.

3.4.1 Summary

In in order to integrate existing legacy software components into distributed architectures several strategies may be considered:

- reengineering of the software components using recent technologies;
- adaptation of existing software components;
- development of wrappers that encapsulate existing software components and provided additional required capabilities.

The most versatile and easy to use approach in the context of systems for symbolic computation is to develop wrapper components that act as mediators between clients and computational engines.

Black box encapsulation is preferable to white box encapsulation because the level of insight regarding encapsulated software components is lower and encapsulation is easier to apply if the range of systems to encapsulate is large. To favour interoperability the wrapper should be designed to use a single generic data model in interaction with its clients and internally translate data from the generic data model format to the one required by the encapsulated software component.

3.5 Suitable Distributed Technologies for Symbolic Computing

The computational infrastructure that we intend to use for large scale symbolic computation problems is heterogeneous and highly dynamic. The computational resources required by large symbolic computation problems can be obtained by bringing together geographically scattered resources provided by research institutions and universities willing to share their computing power. Although such institutions are willing to share their

resources, their already established computational domains and enforced rules cannot be easily changed. Any system that wants to use such resources must be versatile enough to cope with specific particularities of individual computational nodes and their respective computational domains. Appropriate communication technologies, rules to be enforced system-wide and software tools have to be carefully selected to ensure compatibility with the computational infrastructure they want to build upon.

The early distributed systems for symbolic computations that were built had to rely on existing technologies available at the time such as RMI [9] or CORBA [183]. Their primary goal was to provide small to medium scale systems that would usually use hardware resources provided by the local computational domain. One such example is the framework described in [159]. Even though they had as a target to create systems that rely on distributed computational resources, both MathWeb [94] and JavaMath [170] have the disadvantage to use technologies that are not viable for systems that spread over multiple computational domains. The first impediment is that they are not open enough to allow clients and service providers to choose their platform and programming languages they prefer for building clients and services. RMI is even more restrictive than CORBA in this respect.

Another important limitation that systems build using CORBA and RMI have is that they often require more permissive security policies to be implemented by domain firewalls. Since security threats represent a major concern in current systems, it is often the case that administrative rules prevent these systems to function correctly. Limitations of the RMI and CORBA motivated researchers and system developers to find more versatile solutions to implement distributed systems. As a result, Web Services were created and widely adopted as a compromise between interoperability and security on one hand and system efficiency on the other. The underlying architectural style that Web Services are based on is the routine-subroutine style and therefore mappings between service operations and functions provided by CASs are easy to achieve.

Grid technologies, which were initially designed to use TCP/IP socket connections for

communication, have also evolved to adopt Web Services. As identified in [147], the use of Grid services for building distributed infrastructures for symbolic computations may be beneficial in several respects. Amongst them, the WSRF frameworks could be used to implement Multilateral Simple Conversation patterns for which WS-Resources mechanisms provides automatic state support [83]. With the use of WSRF a service becomes stateful and a returning client is automatically recognized and session data can be retrieved from the associated WS-Resource. Additionally, automatic resource management may be used to free resources, a similar functionality with the one provide by the Java garbage collector.

While we consider these features to be helpful, we believe that there are several other features that are even more important for symbolic computing than the ones mentioned so far. Grid services have native support for security which eliminates the burden of enforcing security and designing appropriate security policies over disparate computational domains. Another important benefit is that Grid services provide data management capabilities. Dedicated interfaces and protocols provide secure, reliable and easy to use solutions for moving large sets of data from one computational node to another. Through these services they ease the process of integrating disparate computational resources into a coherent whole. The advantages that Grid services provide for scientific computations in general and their direct support for the requirements discussed in Section 3.2 qualify Grid technologies to be used for symbolic computations.

The CAS Server components were therefore designed to use the capabilities that Grid services have to offer. Execution, data management and discovery services that the CAS Server interface has to provide were implemented using WSRF compliant Grid Services. CAS Server uses specific features of WSRF where they were required whereas generality of the solution was kept whenever possible due to rapid evolution of technologies that may require that CAS Servers have to accommodate new standards and technologies. We found the WSRF mechanisms to be particularly useful for describing the symbolic capabilities that the CAS Server provides to its clients through its interface. Information about the CASs that the CAS Server encapsulates and the functions that are available

for remote invocations are organized as a WS-Resource. Native indexing capabilities of Grids can therefore be used to discover these details. While our discovery process does not rely on the native provided functionality, these capabilities may be useful for compatibility with other systems.

Using Grid or Web services to expose functionality of CASs may also have small impediments. One such example is the lack of support for exposing more than one operation with the same name and with different argument lists. This limitation comes from the standard the WSDL 2.0 [1] which explicitly forbids that operations with the same name exist within the same service definition. This is not the case with regular CASs which may provide functions that have the same name but with a different type and number of parameters. Therefore one-to-one correspondence between a CAS function and an operation on the interface of the CAS Server would not be possible. Even if such restrictions did not exist, it is still not convenient to have services exposing thousands of operations as we would be forced to provide if one-to-one correspondence were to be used. The experience gained by constructing the Computer Algebra to Grid Services (CAGS) tool [60] has let us to the conclusion that the better approach is to use a single operation through which task requests should be submitted.

This design has the advantage to provide a static and standard set of that the client may use in a dynamic way. If new functions are implemented at the CAS level and the administrator exposes them as new accepted operations accessible to remote clients, the interface of the service does not need to change. It is only necessary that the function is registered in the internal Local Registry of the CAS Server. Registration of new functions is the only deployment step required. It is not necessary to recompile or restart the Grid service as is needed in the case of GENSS services which require that a new Java operation is implemented for every new CAS function exposed.

3.6 CAS Level Message Encoding

A key aspect to consider for CAS to CAS communication is the encoding used for data exchange among different CASs. Interaction between a remote client and the CAS engine has to rely on a data model that is understood by both communication parties and ensures that the content of the message is strictly determined. Semantic information is required in the case of symbolic computation to ensure that messages formulated by one system have the exact same meaning after they are decoded at the other end of the communication channel.

As described in Subsection 2.6.1 OpenMath is one of the best choices for encoding mathematical formulae due to the semantic annotations that it provides. Mapping between mathematical content formulated using OpenMath and the internal data model used by a CAS is provided by translators called phrasebooks [157]. Several CASs such as AXIOM [26], GAP [3], Mathematica [24] have implemented phrasebooks that provide support for a wide range of mathematical concepts while for other CASs such components are under development. Due to its features and related software tools that exist for OpenMath, we also consider it as the main solution for encoding mathematical content.

We therefore rely on OpenMath as the encoding standard of messages that describe the tasks request formulated at client side and we implement the required parsers to decode the information at the CAS Server level if such parsers do not exist. Depending on the level of support that CAS engines offer for OpenMath, CAS Server can be used with two types of encodings. One type relies exclusively on the OpenMath encoding for all details that describe the task, while the other adheres to the OpenMath encoding to a certain extent. If the second model is used, mathematical content is not entirely encoded using OpenMath. A predefined OpenMath structure is used as a container for plain string representations of formulae that are specific to a particular CAS.

The SOAP messages that are exchanged between a client and a Web service only represent a container for the messages that are intended to be understood by CASs. The

actual messages that are forwarded to the CAS are received by the CAS Server as flattened XML representations and they are transformed in XML format or plain commands format before they are sent to the CAS. Either of the two message encodings aforementioned could be used, the preferred one being the full OpenMath encoding.

Systems for symbolic computations have used OpenMath as the best choice to encode mathematical content even before CASs were able to understand OpenMath. Various systems have used it to send mathematical content between communicating parties. For MathWeb for instance, mediator components translate OpenMath objects in actual calls specific to Maple, Magma and GAP. JavaMath uses OpenMath as the data encoding standard for sending computational requests but plain string encodings are also allowed. More recently, projects such as Monet, MathBroker and GENSS use OpenMath not only to encode request and responses but also to describe the interfaces of the services they provide. Matching algorithms implemented by brokers use OpenMath encodings to search for appropriate services that could be used to solve a given problem.

3.6.1 Encoding with OpenMath and SCSCP

One of the goals of the SCIEnce project was to develop a communication protocol that would enable CASs to interact using a standard data encoding model. As a result SCSCP [96] protocol was designed. The SCSCP has become a de-facto standard with implementations available for many CASs. Several major CASs, amongst them GAP and Maple, Kant, Macaulay [109], Mathematica, MuPAD, TRIP [103] provide support for SCSCP. Frameworks and libraries for SCSCP implementations are available in C/C++ [16] and Java [17].

The design and the implementation of CAS Server and the design of the SCSCP protocol were done by two distinct teams working in the framework of the SCIEnce project and the CAS Server component was one of the first to support the use of SCSCP. As further described in Subsection 3.6.2 CAS Server supports a second format for encoding data.

There are two dimensions of the SCSCP protocol that influence CAS Server's design. The first one is related to the message encoding. It specifies the possible request and response formats for messages that a client exchanges with the SCSCP enables server. Secondly, the SCSCP protocol encourages CASs to act as service providers. Playing the server role, a CAS should be started as a daemon process that listens to specific TCP/IP ports to which requests formulated using the SCSCP protocol should be submitted.

Even if the CAS is not prepared to provide TCP/IP connections this should not represent a major impediment. Its ability to understand SCSCP would still represent an important step ahead towards interoperability with other CASs. Alternative means could be used to deliver the messages to the CAS and retrieve the responses. According to SCSCP specification, any message exchanged between CASs should be a valid OpenMath object describing the call and meta-data regarding the call. Therefore, the CAS should also implement the OpenMath CDs used by the client to formulate the request. Currently, the support for OpenMath is growing and an increasing number of CASs consider implementing OpenMath parsers.

The SCSCP calls target functions that are implemented by the CAS handling the call. When parsing a SCSCP call, the CAS should be able to identify the function that internally should be executed and the list of arguments that have to be passed. Basically, OpenMath symbols from the SCSCP call are mapped locally to function names. By placing a certain OpenMath symbol inside the call the message actually requests that the associated local function is invoked. All arguments specified within the call and all responses should be described using OpenMath standard. An example of such message is given in Listing 3.1.

```
1.    <OMOBJ>
2.        <OMATTR>
3.            <OMATP>
4.                <OMS cd="scscp1" name="call_ID"/>
5.                <OMSTR>anid</OMSTR>
6.            </OMATP>
7.            <OMA>
8.                <OMS cd="scscp1" name="procedure_call"/>
9.                <OMA>
10.                   <OMS cd="SCSCP_transient_1" name="Factorial"/>
11.                   <OMI> 10</OMI>
12.               </OMA>
13.           </OMA>
14.       </OMATTR>
15.   </OMOBJ>
```

Listing 3.1: Example of SCSCP Call

The call in Listing 3.1 represents a simple example that requests a the computation of a factorial. The header section of the SCSCP message may specify meta information regarding the request and the computational requirements that the machine on which the CAS is running should meet to be able to handle the call. Within the call the header is specified using the <OMATP> element starting at *line 3*. Conversational communication patterns may even be supported by using a cookie mechanism that is able to relate multiple calls to a single client session. The mechanism of cookies that a CAS is able to understand should be supported by the the inner core of the CAS. External mechanism that could provide support for this feature, such as WS-Resources, are less generic. Internal management should be preferred when sessions are required.

The OpenMath symbol used at *line 8* is specific to SCSCP and instructs the CAS parsing the call that this is a remote call that targets a function implemented by the CAS. At *line 10* the message specifies the OpenMath symbol that identifies the function that should be called, and further, it states that the simple OpenMath object <OMI>10</OMI> should

be passed as a parameter. Based on its internal configurations the CAS should be able to identify the correct function to call internally, to execute it and formulate a response to be returned to CAS's client.

Using OpenMath for data encoding is an important step forward for CAS to CAS interoperability. The use of OpenMath ensures that both the request and the response provide sufficient information to be mapped to internal data types in a deterministic way. Any CAS that implements support for the OpenMath dictionaries used within the call is able to understand the call and to take the appropriate actions. The SCSCP protocol provides a clear message structure that should be preferred for CAS to CAS communication.

3.6.2 Encoding with OpenMath and Plain Text

Most of the CASs do not yet support OpenMath as an encoding model for data exchange with other CASs. Older versions of CASs that do not support OpenMath are still in use and a migration process is not entirely possible due to compatibility issues between older and newer versions. Non-standard data representations meaningful only for a certain CAS or even for a certain version of a CAS are therefore still required. Integration of such CASs within distributed environments is also necessary due to the functionality that these CASs provide.

The same generic *execute()* operation provided by the CAS Server's interface as single point of entry can be called using two types of encodings to describe the task. Additional to the format specified by SCSCP protocol tasks can be encoded as surrogate OpenMath objects. This alternative encoding uses OpenMath as a frame in which various details regarding which CAS engine, which function from which package should be invoked and which are the arguments to be passed to the function call. The code snippet shown below provides a generic example of this format.

At client side a remote function call is translated to the corresponding OpenMath object as the one in the following example. The message is parsed at CAS Server side and the

information encapsulated in the OpenMath object is used to create the appropriate CAS specific command. This process is similar to the one that phrasebooks use to translate OpenMath encoded objects to commands that a particular CAS understands. The full OpenMath encoding is preferred because it is more generic and any CAS implementing a particular OpenMath CD internally maps OpenMath objects to data structure. This is not the case with the encoding below since the *procedure*, *package*, and argument details are specific for a certain CAS and are meaningless when used with other CASs.

```
<OMOBJ>
    <OMA>
        <OMS cd="casall1" name="procedure_call"/>
        <OMSTR>procedure</OMSTR>
        <OMSTR>package</OMSTR>
        <OMSTR>Arg1</OMSTR>
        <OMSTR>Arg2</OMSTR>
    </OMA>
<OMOBJ>
```

Listing 3.2: Example of Plain Call Encoding

In Listing 3.2 the *procedure_call* OpenMath symbol marks the type of call being formulated. The first two *OMSTR* objects describe the function to be called and the package that the function is part of. The rest of the following OpenMath string objects, in our case *Arg1, Arg2* represent the plain string encodings of the arguments that have to be passed to the function call.

Using the message encoding in Listing 3.2 functions implemented by CASs are made available through remote function invocation. This approach though breaks the CAS to CAS interoperability requirement and it should be used only as a compromise for CASs that do not support SCSCP and OpenMath. Another problem is that arguments are not encoded using a standard format and therefore the function to which the arguments are passed has to implement ad-hoc functionality to parse and interpret the string representations.

Regardless of the data type of the arguments at the client side, the remote function call will only receive their plain string encodings. For any other type than plain strings, the client must map/transform values to their string representation before enclosing them in the message call. The result obtained by calling the target function is the exact string returned by the CAS, and thus, the client is responsible for parsing the string result and for extracting the useful information.

If for SCSCP format, the Client Manager component of the CAS Server only extracts some meta information and then forwards the original SCSCP encoding to the target CAS. The Client Manager acts like an adapter by implementing a bridge between the client and CAS, through the interface of the Grid Service. The client manager is responsible for extracting the details of the call from the message and formulate a meaningful call that has to be submitted to the CAS. Most often, this requires that a string representation of the call with the format *package.function(Arg1,Arg2)* is created and sent to the CAS to be evaluated. The call string should be exactly the same as the one a human user would submit through the common interface of the CAS that is locally accessible.

An important difference between this type of call and a call submitted through the command line interface of the target CAS is persistence. When working locally with a CAS such as GAP using the command line interface, a function call may affect the state of system or session variables that are stored in the memory of the CAS. A subsequent call in command line could potentially use the initialized values. This interactive behaviour is not available between two subsequent calls to a CAS Server without additional intermediary steps that would store and resume a certain state. For simplicity, calls to the CAS Server must be self explanatory and self contained and no previous state should be assumed.

3.7 Summary

In this chapter we have shown that distributed infrastructures for symbolic computations represent the solution for allowing computer algebra specialists to solve large symbolic problems. The design aspects of the CAS Server component presented in this chapter were previously presented in [58, 61, 148, 150, 129].

Computer Algebra Systems (CASs) represent the main computational engines for symbolic computing. We have shown that the most convenient way to build an infrastructure for symbolic computing is to reuse the capabilities of CASs by integrating them in a broader architecture. There are three important problems that have to be considered for successful integration of CASs: the encapsulation technique used to communicate with the CAS that further allows remote clients to communicate with the CAS; the data model used for encoding messages exchanged by the client and the CAS; the technology to be used for exposing CASs functionality to ensure that potential clients may access the functionality in a seamless fashion.

The CAS Server component was designed to allow more than one CAS to be exposed through the same interface. To achieve this, the CAS Server acts like a mediator between remote clients and exposed CASs. As discussed in Section 3.4 the way a CAS may be interconnected with the CAS Server depends on the capabilities that the CAS natively implements. Building wrappers specific to a certain CAS is the most convenient and flexible solution.

Interoperability represents one of the major issues in establishing a distributed symbolic environment. Lack of interoperability impedes potential clients from accessing functionality provided by CAS Servers. To overcome these problems three important issues have to be addressed: the consistency of the interfaces; the data model used for encoding messages and; the technology used for implementing the interfaces. The structure of the interface that the CAS Server exposes is unchanged irrespective the CASs that are exposed by the CAS Server. Driven by the requirements specified in 3.2, as a minimum

the following set of capabilities has to be provided: the ability to receive computational tasks and provide the results in an asynchronous way; a single point of entry through which the tasks should be submitted allowing thus more than one CAS to be exposed through the same interface; a set of operations to allow the user to discover provided functionality and; capabilities to manage execution of tasks. We have also shown that exposing CASs functionality should be done by permitting clients to access selected routines implemented by various CASs.

Symbolic components integrated in a distributed architecture, whether they are service providers or clients must use a common encoding format that can be used by all parties. The most successful standard for encoding mathematical formulae is OpenMath. The SCSCP protocol is currently the 'de facto' standard for CAS to CAS communication. We have designed the CAS Server component to use the SCSCP protocol for interconnection with CASs as most of the popular CASs already provide support of SCSCP. The use of OpenMath and SCSCP protocol is of paramount importance for interoperability and cooperation between CASs. Lack of support for SCSCP and OpenMath may be partially overcome by using an alternative non-standard encoding model that we have described in Subsection 3.6.2. The latter has the advantage that it may be used as a workaround in particular scenarios but it lacks the generality and flexibility that SCSCP and OpenMath provide.

Due to the advantages that Grid Services provides, as shown in 3.5 Grids may be considered as the most suitable technology of building a distributed infrastructure for symbolic computations. We have used WSRF compliant Grid Services to implement the CAS Server component's interface. Thus, the CAS Server may provide access to one or more CASs installed on a ordinary desktop machine or, in more advanced set-ups, it may hide a whole LAN or computational cluster.

The CAS Server is therefore a suitable solution for exposing CAS functionality to be accessible by remote clients. Grid Services and Web Services are standardized solutions for implementing the RPC architectural style. The interface of services is clearly defined

by the WSDL document of the service and any client can use a service if they are able to formulate correct requests, independent of the platform they use. New functions implemented by a CAS can be easily exposed through the Grid Service interface without the need to modify the services. In addition CAS Server provides a set of functionality that allows clients to discover which functions the CAS Server provides and they can control the execution of tasks by pausing, resuming and cancelling tasks. The advantages that Grid Services provide in comparison with Web Services for implementing CAS Server are default security mechanisms and data management services that allow seamless transfer of files between execution nodes.

Chapter 4

Orchestration of Web/Grid Symbolic Services

This chapter addresses the problem of composing the functionality of several CASs for solving symbolic computation problems as reported in [60, 61, 66, 148]. In Section 4.1 we analyse scientific workflows particularities and the special requirements they raise. Symbolic computation workflows have to be expressed in a format that can be understood by existing workflow execution engines, usually as compositions of generic workflow patterns. In Section 4.1.2 we provide a set of guidelines for translating common existing patterns in symbolic computations to the generic workflow pattern format.

An overview of generic tools and technologies for description, execution and management of scientific workflows is provided in Section 4.3. Their capabilities may be used to support the execution of symbolic computation workflows. In Section 4.4 we introduce a new component of our architecture, namely the Architecture for Grid Symbolic Services Orchestration (AGSSO) Server component. The AGSSO Server provides support for automatic execution of workflows for symbolic computation by orchestrating CAS Server components previously described in Chapter 3.

4.1 Service Orchestration for Symbolic Computing

In Subsection 4.1.1 we analyse the most important differences between regular business workflows and workflows for scientific computation. In Subsection 4.1.2 we briefly present the most common workflow patterns and workflow categories.

4.1.1 Scientific Workflows and Their Requirements

The emergence of Web and Grid standards and the democratization of access to computing power have had a major impact on the way scientific research results are obtained and disseminated. The main motivation for creating a distributed computing architecture regardless of the actual set of technologies used is to deliver the computing power and software tools needed to solve large scientific problems. Powerful computing tools enable scientists to share their results and to test hypotheses in a seamless fashion.

Scientific computations and experiments are different from running simple and isolated computational tasks because they may involve thousands of execution steps and require access to data and computational capabilities situated at geographically scattered locations. This is also the case for large symbolic computing problems which require on one hand large computational resources and on the other hand specialized software that may not be available on a single computational site. In order to solve large symbolic problems computational resources and capabilities that can be offered by a single machine or a LAN may not suffice.

The most simple and straightforward composition of CASs' functionality may be obtained just by accessing through the usual user interface of a CAS the capabilities of another CAS instance. Most of the existing systems that implement capabilities for CAS to CAS collaboration such as MathWeb-SB [94] provide support for simple composition. This type of composition is suitable for less complex computational problems and for problems for which user steering is required between individual calls. For problems

that can be run in batch mode, i.e. no steering is required, a better solution is to provide mechanisms and software infrastructure for automatic execution management.

Depending on the data access patterns specific to a certain problem, even if parallel versions of algorithms may be used to solve a problem, it is not always efficient to use highly distributed infrastructures. One such example is the computation of Gröbner Basis [123] for which current algorithms are not suited for massive distributed environment due to close data dependencies between individual steps of the algorithm [33]. For other problem classes, such as the ones similar to the orbit enumeration algorithm [126], solutions can be implemented by combining independent services [66].

Decomposition of an initial problem in smaller problems and solving them using a distributed computational infrastructure may represent on one hand a solution to provide the required computational power and storage capabilities and on the other hand may significantly reduce the wall clock time required to solve the problem. As a result of the decomposition process a sent of individual execution steps are identified. The interdependency between these steps has to be thoroughly documented to ensure correct management of the execution. A high level view of the computation is required during planning and execution phases. The most common way to represent such processes, generically named workflows, is through directed acyclic graphs (DAG) which describe an abstract representation of the computation. The nodes of the graph identify computational steps and arcs determine the dependency between computational steps. Dependency relations may be due to data dependencies that one task generates which is required as input by another task or they may just be required by the impact that certain tasks have on the overall state of the system.

Management of workflows with a low level of complexity is feasible even if it is done by the client components. In this case the client has to explicitly send requests for each node representing a computational step, collect results and formulate the subsequent requests until the computation is finished. This solution is also suitable for client side applications which occasionally require access to functionality that the application does

not natively provide but is made available by external services. The client centred execution approach may be easy to use for small scale problems by users that are comfortable with using technologies and programming languages that provide support for interacting with external services. The overall complexity of the system may also be smaller since support components that should assist the users in automatic workflow management, such as components for service discovery, scheduling and load balancing are not required. In this cases, the client itself is responsible for managing the whole computation, including discovery of suitable services. As shown in Subsection 3.2 this approach has several shortcomings related to usability and performance of the whole system and form complex workflows and more advanced solution should be used.

The effort required to maintain verbose workflow descriptions can also be significant. Hard coded implementations of service compositions are difficult to understand because they are cluttered with explicit calls and error handling code sections. They are also hard to adapt and maintain. The overall evolution of the system is impeded since every change that occurs in the interface or in the location of services has to be reflected in the source code combining the services [130]. Describing workflows in more abstract terms that do not contain low level details about the actual services to be used is a more flexible solution especially for describing large and complicated workflows. Workflow execution is best achieved by specialized management engines that are able to follow the execution process and provide built in capabilities for fault management and compensation handlers.

Due to the nature of the problems they have to solve, environments for scientific computing must provide support for features that are less common for business oriented architectures. The structure and the order of magnitude of scientific workflows is much different than the ones of business oriented applications. In business environments, tasks usually require a short time to complete and the number of the tasks composing a workflow is relatively small while scientific workflows are composed of thousand of steps and each individual step may need a long time to complete.

The main requirements that a system for management of scientific workflows should meet include capabilities to combine components in a modular way, exception handling and compensation mechanisms, and management capabilities [34]. Usually, most of these capabilities are provided by centralized workflow managers generically called workflow execution engines. The role of the execution engine is to provide the necessary mechanisms to bind, invoke and retrieve results from external services according to the description of the deployed workflow. The most important requirements raised by scientific applications that such engines should provide were previously analysed in [105, 166]. These can be grouped into three important categories:

1. Integration with existing standards and software technologies,
2. Service discovery and workflow management capabilities,
3. Runtime requirements.

Workflow engines should be flexible enough to the adapt to vast variety of technologies that are used to implement services. Even if most existing services are provided using the Web Services standards, such engines must be able to interact with services that are implemented using technologies such as RMI or CORBA. Usually, out of the box engines provide native capabilities for interacting with Web Services while for other technologies, extension interfaces are defined. This is also the case for the ActiveBPEL [28] execution engine, one of the most popular execution engines, which we have integrated as part of our AGSSO Server component. To assist the user in the process of service discovery and workflows execution, the engine should be capable of interrogating external discovery registries from which to retrieve the addresses of the services to be used.

Workflow management capabilities are extremely important in the context of scientific computing. Because the workflows as a whole and the individual tasks that the workflow is composed of may require long time to compute, it is important to have the ability to monitor the tasks and steer their execution when necessary. Tasks that take too long

to be computed could be discarded and alternative solutions applied if the human expert chooses so. Task completion may require a large amount of computational resources and it is sometimes required to temporarily free resources by pausing and even cancelling a task. If check-pointing is available, long running tasks may survive computational disruptions caused by system failures or regular maintenance activities such as system updates or restarts.

Monitoring capabilities should allow users to supervise workflows' execution. If execution errors occur, the system should be able to handle the errors and even to cancel the whole execution process and to restore the system to its previous state. During execution, the user should be able to manage the execution by pausing or cancelling it. Intermediate results should be stored by the system for later reference and to allow the system to resume computation in case of failure. Errors may also occur due to non standard interfaces of composed services and the data types they use. The workflow engine has therefore to provide mechanisms that are flexible enough to handle such cases.

On occasions, the human expert may even guess which are the expected results for one or more tasks that are part of a complex workflow. Therefore the specialist can choose to override a particular task by assigning a specific value that the user wants to consider for the given task. On one hand this feature may dramatically improve the overall time of the computation because the individual tasks for which the result is assumed and not computed are skipped. On the other hand, based on the same capability, it is even possible for a scientist to experiment with different values manually assigned to different tasks without actually changing the computational steps. The specialist can thus investigate possible results that can be obtained by running the same workflow for multiples cases and testing possible results obtained for when partial results are manually assigned.

Scientific discovery is only valid if the obtained results can be replicated and the way that they were obtained properly documented. Reusability and reproducibility are particularly important in scientific processes and a workflow management system should save all relevant meta information needed to support these requirements [105]. Exam-

ples of data that should be stored include the abstract workflow description, job requests and obtained results, details regarding the configuration of the execution nodes that execute the jobs of the workflow and execution duration. Execution engines store automatically some of the required data but additional features are often required to be implemented. Once results of complicated or long running processes are obtained, they should be stored and reused if possible.

Workflow description languages like WSFL or BPEL that are designed to be used for describing workflows in format suitable for processing by workflow engines are complicated and contain details that are not of immediate interest for application experts. An overview of such languages, workflow engines and techniques used for automatic composition of Web Services is given in [171, 146]. Application experts should use appropriate high level languages, relevant and intuitive for a specific application domain, to describe their workflows and not complicated languages that workflow engines understand [105]. High level languages must provide capabilities to compose services in a seamless way while low level details are hidden. The actual services' details involved in workflows' execution can be filled-in at run time by support components. Auxiliary management steps that cover transfer of required data or specific steps that should be executed to interact with a specific type of service can also be automatically provided [166].

The design of the workflow description languages for scientific workflows have to be appropriate to support the way scientists interact with computing infrastructure. Clients communicate with servers using massages that describe the actual request and not how the goals should be achieved, therefore they are rather descriptive than instructive [34]. Consequently, the users should be able to invoke already deployed services rather than defining themselves the code that needs to be executed to achieve a given goal. When there is a need to combine several services to achieve a goal, high level constructs that define the workflow should be provided to the user.

Current workflow execution engines are able to partially support the requirements men-

tioned above either directly through auxiliary add-ons but none of the existing solutions is able to cover all the requirements stated in a way such that they are straightforward to use. Additional components that extend the functionality provided by these engines have to be implemented and integrated. Solutions that partially exist for particular domains cannot be adopted without applying changes that make them suitable for the targeted domain. One simple example is the use of visual tools to compose and manage workflows which may be appropriate for some domains but impossible to integrate with systems that lack support for visual interfaces.

The requirements that are applicable for scientific workflows in general are also valid for scientific domain of symbolic computations. The features mentioned above would have a positive impact on the development and dissemination of symbolic computation results. A successful solution for scientific workflow management for symbolic computing has therefore to provide support for these features. Within the framework of MONET project, the broker component has the role to discover appropriate services that should be invoked by matching the problem description that a client supplies with the capabilities of existing services. Since more than one client uses the same broker, one of its design requirements was to provide a planning service that should select services based on the general state of the system and problem specific characteristics.

The planner was initially intended to also provide support for automatic composition of services for the case in which a single service was not sufficient to solve a certain problem [32]. Unfortunately, due to the complexity of the problem, a simpler solution was considered [69]. Instead of automatic composition, the system was designed to provide mechanisms that allow administrators of the system to deploy BPEL workflows that combine existing services. They proved the feasibility of the solution for services that could be executed in sequence. The disadvantage of the solution they proposed is that regular users were not able to describe and deploy their own workflows, and therefore they were restricted to use only existing ones.

The Distributed Maple system described in [160] uses scheduler components to select

and connect to Maple instances linked through a Java framework. Handles of tasks may be used as input parameters to other services and therefore dependency relations may be created. Dusher [83] relies on BPEL and WSRF to implement Bilateral Multi-Conversation patterns by calling services that expose GAP. Using this technique, sessions are created to a GAP instance and intermediary results are stored in resource properties of a WS-Resource.

The CAS Server components described in the previous chapter represent the foundations of our architecture. Besides providing means to expose CASs' functionality through Grid Services interfaces, they implement features that can be used to control the way tasks are executed, and provide management and monitoring capabilities, implement mechanisms to store and index for later retrieval the results of computations, provide data management for large data sets. Without these features providing a symbolic infrastructure that meets the requirements mentioned above would not be possible. Our final goal is to provide symbolic researchers a platform that is able to assist them in describing and controlling complicated workflows in a way that it is intuitive but sufficiently complex to cover their needs. The user has to be able to create workflows, send them to be executed by specialized component and retrieve the results of their computations.

To support composition of symbolic services provided by CAS Servers, we have designed and implemented the AGSSO component. This new component we introduce is responsible for receiving computational workflows from clients and to manage them on the behalf of clients. The role of the AGSSO component is to provide on one hand capabilities that are specific to a workflow manager combined with other capabilities such as workflow and task management, service discovery and data manipulation. This component and the functionality it provides are further discussed in Section 4.4.

4.1.2 Workflows and Workflow Patterns

The lifecycle of a workflow is composed of several stages that are determined by the level of detail known about its structure, services to be invoked and arguments that should be

provided to match the parameters expected by services [76]:

1. Workflow as a template - general problems are described as workflows using constructs specific to a certain computation domain; the role of a template workflow is to describe in general terms the computational steps that have to be undertaken to solve a class of problems

2. Workflow instance - is obtained from a template workflow by supplying the arguments matching the parameters expected by the workflow as input so it may be executed;

3. Executable workflow - based on a workflow instance the workflow manager binds the task of the workflow on specific computational resources

Depending on the architecture of the systems and the role various components have in the workflow's life cycle, the workflow could evolve from the template stage to the executable stage within the boundaries of a single component, or different components may participate to its life cycle. For example, Active BPEL Designer could be used to describe the workflow template, to deploy it and to invoke it by providing arguments to the resulting Web Service. More often though, the border line between the stages is not obvious and more than one component collaborate on the workflow's path from template to executable.

Scientific platforms for workflow management decouple the stages of the workflow in even more fine grained steps. The description of the workflow takes place at the client component level where the user describes the workflow's structure by combining abstract computational constructs. At this level, constructs that are familiar to the user level of expertise are used and based on the user actions, a workflow encoded using a generic workflow language is created. Once the template workflow is described, the client component submits the workflow to a specialized workflow management component which translates the workflow into a workflow language specific to one of the available work-

flow execution engines that the system uses. The resulting workflow is still a template since specific input values are not specified yet.

As soon as the user specifies the input parameters, the workflow becomes an instance. The stage of the workflow in which system determines the actual resources to be used for execution influences the efficiency of the composition process and differentiates the composition technique to be used. Available resources' configurations are highly dynamic and heterogeneous. Over time new services are implemented while older ones vanish or change their interface or the functionality they provide. Static composition is based on existing services of which details are known at the workflow's design time and they cannot be replaced based on the state of the system at a given time. This type of composition has the advantage that the services are known a priori and therefore compatibility problems can be avoided. As a consequence, it is also the less flexible composition technique because once a service is modified or no longer available the workflow itself has to be altered or it will fail to execute.

Slightly more versatile is static composition with dynamic bindings. The structure of the services to be used may be assumed at design time while the actual location of the services may be determined based on the latest available information just before submitting a task to be computed. This type of composition is possible if the technology used for implementing the services permits decoupling of the description of the service's interface from the actual location where the service resides. In this case, the workflow is static with regard to the interfaces but it remains valid if a service having the same interface but hosted by another component of the architecture is chosen. Using this type of composition may improve efficiency because the binding to the actual services to use is deferred until the service needs to be invoked. The best services may be selected based on the current state and load of the system.

The most versatile but also the most error prone type of composition is the one in which the actual services used to solve a problem are discovered at runtime. Typically, the applications specialist describes a problem that needs to be solved and depending on

the nature of the problem and input parameters the system is able to determine in an automatic way which services have to be combined. To achieve this goal, the workflow engine has to be able to understand the problem and to be able to match requirements onto capabilities that existing services can provide. Clearly, the above solution is the most versatile but due to lack of standards and insufficient support currently it can only be applied in specific domains. A universal solution for dynamic composition represents a desideratum still to be attained.

The primary role of a workflow management component is to combine the functionality implemented by other software components, regardless the the technology used for communication. Workflow execution patterns represent the building blocks that the user can combine to describe a compound computation. Execution patterns have the advantage of allowing the user to describe the solution at a high level of abstraction for which only details that are of immediate concern of the user are specified. Usual control flow constructs that a programmer uses to implement algorithms have corresponding counterparts to be used for describing workflows.

Workflow patterns may also be used to evaluate the expressiveness and suitability of different languages and composition techniques used for expressing workflows [116, 27]. The workflow patterns are particularly important for describing the nature of interactions that occur in distributed environments between autonomous components that are orchestrated towards a common goal. The number of patterns that apply to Web service composition is quite large and they try to capture behavioural nuances that may occur. Several patterns though represent the foundations on which the other patterns rely upon. Basic patterns identified in [139] were further used to investigate the expressiveness of existing workflow languages [198]. A short overview of the most common workflow patterns is presented below.

A common pattern, the *sequence pattern* represents the sequential execution of two or more tasks. The dependency between certain steps may be purely functional or imposed by data dependencies that exist between these tasks. Due to the dependency amongst

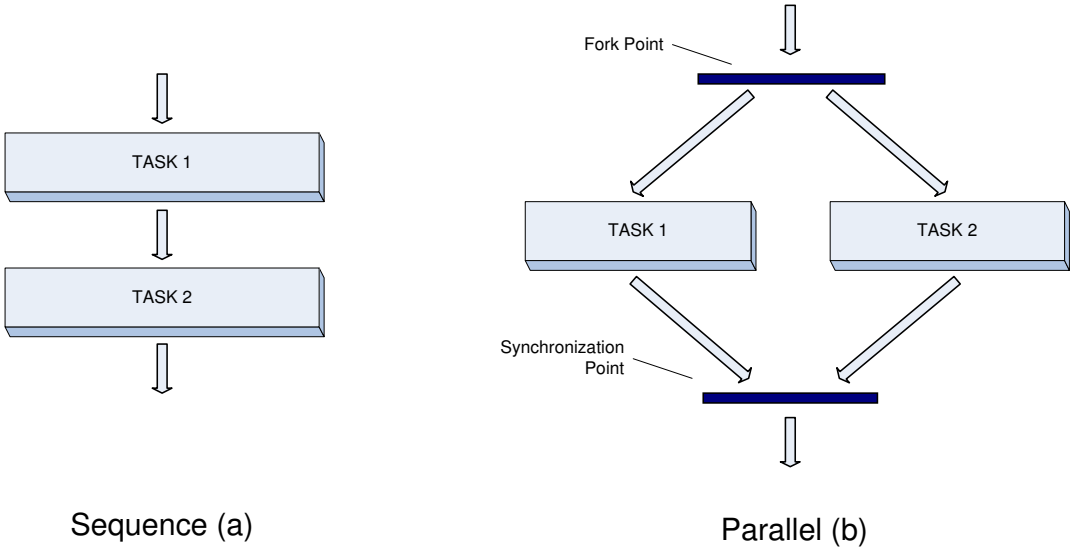


Figure 4.1: Sequence and Parallel Execution

them, the tasks have to executed one after another and a dependent task cannot be executed unless all the tasks it depends on are finished. As shown in Figure 4.1(a), *Task 2* must wait for *Task 1* to complete before it can be processed.

If there is no dependency among tasks, they may be executed in parallel as a *parallel split pattern* that describes a process fork. If the subprocesses reunite at a certain moment of the execution, that point is a join point and the parallel split is with synchronization. As shown in Figure 4.1, the two tasks have no interdependency and after the two are completed the processing branches reunite. This pattern assumes that every branch is executed only once. As a variation of this pattern, the *multiple instances without synchronization* pattern occurs when multiple instances of the same task must be executed in parallel but no synchronization is required after they complete.

A task or a a group of tasks may have to be executed only if a condition is met. Conventional programming languages provide the conditional construct *if-then-else*. Such behaviour may be expressed using *conditional patterns* depicted in Figure 4.2. The *exclusive choice pattern* selects, amongst several possible branches, the branch that should

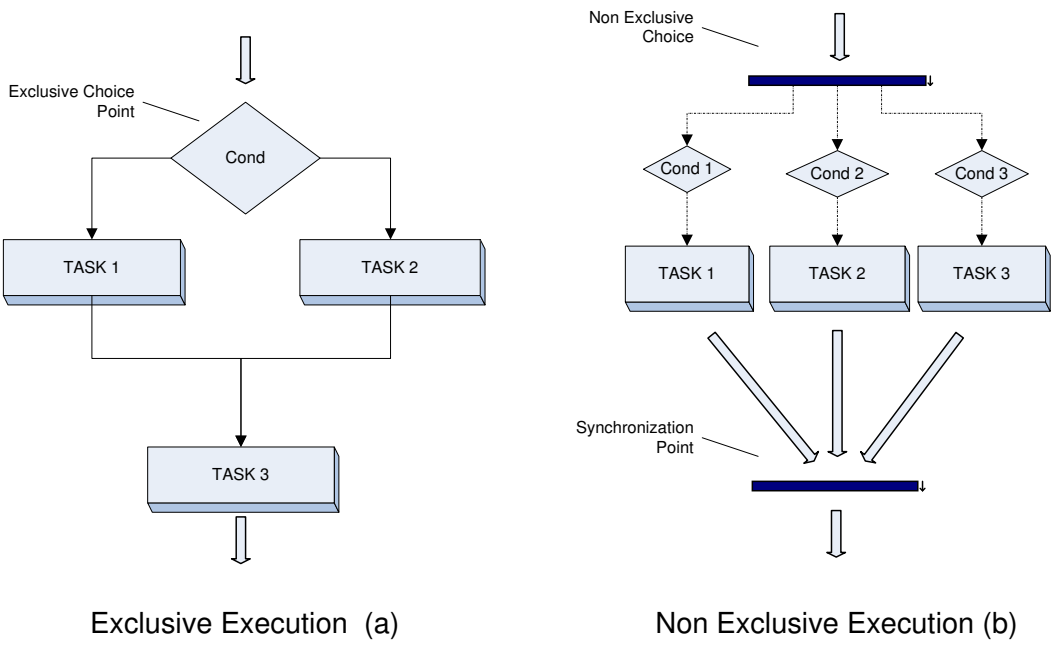


Figure 4.2: Conditional Execution Patterns

be executed based on the evaluated condition. Because only one branch can be activated at a time the simple merge point that reunite the branches in Figure 4.2(a) is not a synchronization point. Similarly, the *multichoice pattern* 4.2(b), uses conditions to determine if an execution branch should be activated or not. Unlike the first conditional pattern, this pattern allows several branches to be simultaneously activated. For all branches for which the corresponding condition is met the execution starts and the tasks are executed in parallel.

One can potentially identify more than one possible approach that could be used for solving a problem and may want to try them by executing them in parallel. The approach that provides the fastest answer is considered and all the rest of the executions that were started and not yet completed are aborted. Several solving algorithms and techniques are therefore tested at the same time by concurrent processes and, as soon as one solution is obtained, the rest of the processes may be discarded. This execution pattern, *deferred choice pattern* depicted in Figure 4.3(a), is particularly useful for symbolic computations. It is often the case that the computer algebra specialist may use multiple

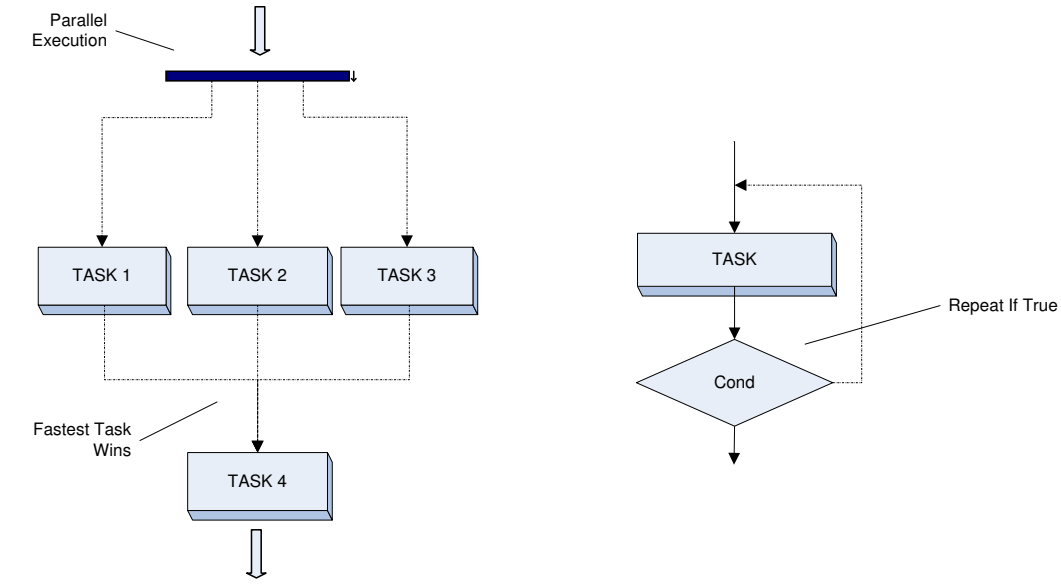


Figure 4.3: Deferred and Repetitive Patterns

algorithms to solve the same problem but it is difficult to predict which is the one that will compute first.

Often there are situations when the same action must be executed several times with various input arguments where the number of iterations is known in advance. Conventional programming languages implement this construct as *for <codition> do...* or *while <condition> do...* loops. This behaviour is described as the *multiple instances with prior knowledge pattern*. A variation of this pattern depicted in Figure. 4.3(b) is the *multiple instances without prior knowledge* when an external factor that cannot be anticipated determines the end of the loop execution. This pattern is also supported by conventional programming languages in the form of *repeat...until <condition>* constructs. The number of iterations that must be executed when the second pattern is used is determined by the processing itself. An example that fits this pattern is processing of a list of objects to which new objects can be dynamically added during the execution. For this type of problem the total number of objects that have to be processed is not known

when the actual processing starts. The processing ends only when there are no more objects in the list to be processed.

Virtually any computation algorithm can be described using the execution patterns described above. The task blocks from the diagrams above can represent atomic tasks or they may be replaced with other patterns. Therefore, symbolic execution patterns that are used in symbolic processing can also be described at abstract level using the above patterns. Workflow engines that support these fundamental patterns may also be used to execute workflows for symbolic computations with the remark that any symbolic evaluation has to be done by a specialized component since workflow engines do not provide support for symbolic computations.

4.1.3 Summary

The main characteristics that differentiate scientific workflows from business workflows are their significant large size, the long time they require for execution and large number of task generated by iterative processing. Symbolic computation workflows have similar characteristics with other scientific workflows and dedicated workflow execution managers are the most efficient way to execute such workflows.

To be executed by workflow managers, workflows have to be expressed using special languages that are suited for automated processing but verbose and complex. These languages are not suited to be used by human users directly and therefore more intuitive, clear and concise constructs that are easy to use by the human experts must be provided. Such languages must provide means to combine existing execution patterns to describe workflows while unnecessary details such as the address of the actual services to be used for execution must be automatically filled in by the supporting system.

In addition to features that are already provided by existing workflow execution engines, scientific workflows require:

- mechanism to support provenance of obtained results;
- workflows re-execution capabilities;
- workflow steering that allow human experts to steer the execution of workflow's execution at runtime;
- workflow execution management such as cancellation, pause and resume of tasks.

Dynamic workflows have the advantage that they can solve problems by combining services that are discovered at runtime. Opposed to dynamic workflows, static workflows can only use services that were indicated at design time. Static composition with dynamic binding represents in-between solution because the structure of the workflow is fixed at design time while services are selected at runtime.

4.2 Basic Patterns in Symbolic Computing

Examples of execution patterns in symbolic computing can be easily drawn from the manipulation capabilities that are offered by existing CASs. Most often, the CASs handle mathematical formulae and structures as objects and list of objects. One of the most developed CAS system is GAP. Using GAP, with a list of objects the user is able to execute several types of operations: apply certain transformations to all the objects of the list, analyse the properties of those objects, create new lists based on certain criteria. Depending on the nature of the problem, manipulations on objects can even be possible in parallel, on remote machines, if the computational gain motivates it.

The control structures used by general algorithms are often part of the standard programming constructs. While their syntax may vary from one CAS to another, systems such as GAP [3], Maple [10], KANT/KASH [21] all provide control structures for control flow and repetitive executions. A peculiarity of those systems is that repetitive constructs are usually available in conjunction with lists of objects. Therefore these systems are not

different with regard to their capabilities to describe arbitrary complicated algorithms, in comparison with popular imperative languages such as Java or C/C++.

Algorithmic solutions of complex problems are obtained through execution of atomic steps in a predefined order. It is the same case for solutions that are specific to symbolic computing. Specific language constructs that control the execution flow within a CAS can be mapped on control flow patterns used to compose Web Services. The actual processing steps requested by a symbolic computing algorithm can also be mapped on Web Services invocations. It is thus possible to translate an implementation of a symbolic algorithm so it can be expressed in terms of workflow patterns and services invocations. Languages that are currently used for describing Web Service workflows are too close to the Web Service orchestration level to be used directly within a CAS. The description of such workflows requires low level details such as the address of the composed Web services and data conversion specification. It is therefore necessary for CASs to provide more abstract and versatile mechanisms to describe such workflows.

By analysing current CASs' capabilities we can identify a mapping of CAS level constructs on more general workflow patterns. Even if it is not always obvious, in fact symbolic computations specialists organize the processing instructions using workflow patterns. Only when dealing with an external workflow execution engine these patterns become more visible.

The simplest execution pattern used in symbolic computing is the sequence pattern. This often arises when the user runs several commands one after another or if function composition is used. Usually, the current state of the system is stored by the command line interface that the application specialist is using. When dealing with external workflow engines, the actual steps of the computation have to be clearly identified and tasks have to be defined explicitly. A hidden sequence pattern implied by a function composition such as:

```
a:= func1(func2(b));
```

must be clearly separated in smaller pieces using explicit sequence markers and task isolation that the workflow engine can translate to invoke calls to external services:

```
sequence {
  c = func2(b)
  a = func1(c)
}
```

Conditional patterns are also allowed in most of the CAS processing languages. The typical form in which they may be expressed is:

```
if ( condition ) {
  //execute true branch statements
}
else {
  //executed false branch statements
}
```

where the CAS is able to evaluate the *condition* specified within the control structure above. In terms of workflow patterns executed by a workflow engine, this construct may be expressed with a small modification:

```
boolean_value = evaluate_condition(condition)
if ( boolean_value ) {
  //execute true branch statements
}
else {
  //executed false branch statements
}
```

Since general purpose workflow engines are not expected to have any capabilities to evaluate symbolic constructs and only simple numerical and boolean evaluation can be used, expressing conditional patterns has to use additional service calls. As it can be

seen in the code above, the evaluation of the condition may not be possible at workflow engine level. The workaround is to use an additional evaluation service that takes the symbolic condition as an input and offers back the result as a boolean. Whenever a conditions has to be evaluated this way a custom service that can do the evaluation for the workflow engine must be used.

Lists represent the main container of objects allowing the CAS to manipulate symbolic objects, usually through repetitive constructs. Any batch processing that may be executed by a CAS is thus related to its capabilities of processing lists. While the list itself is stored within the client machine, the processing of objects composing the list may be done on remote execution nodes. Described in *pseudo-code,* visiting every object contained in a list can be done by applying a repetitive construct such as:

```
for (item in list) {
  //execute transformation on item
}
```

Again, the workflow engine lacks symbolic capabilities and it is not able to understand and evaluate OpenMath objects. All manipulations must be achieved by calling external symbolic services that are able to understand and manipulate the objects. Every step of the repetitive iteration over the elements of a list must be described explicitly by defining tasks that can be executed as remote calls. Here we give an example of a multiple instances with prior knowledge pattern in which the number of elements does not change during execution:

```
end_index = s_size(list)
for (index = 1..end_index) {
  item = get_item_with_index(list,index)
  s_transformation(item)
}
```

In order to map the two constructs mentioned above, additional external services must be invoked: for finding out the *end_index* we need to invoke an external service that will

determine the size of the list; moreover, the object situated at *index* in the list must also be retrieved by accessing an external service. The processing over the object itself has also to be done by an external service.

```
end_index = s_size(list1)
list2 = s_create_empty_list()
for (index = 1..end_index) {
  item = s_get_item(index, list1)
  boolean_value = s_evaluate_object(item)
  if ( boolean_value ) {
    s_store_object(item, list2)
  }
}
list = s_get_list(list2)
```

Listing 4.1: Implementation of Filtered Pattern

The two simple execution patterns mentioned above may be easily combined in order to create more complicated execution scenarios. For instance, suppose we have a list of objects and a selection function that decides a boolean value based on the value of a object. We want to create a new list containing all the objects for which the condition holds. This execution pattern, referred by symbolic computing specialists as the *filtered pattern*, may be achieved by creating a workflow that combines the two patterns above and several pre-existing external services, as shown in Listing 4.1 where all function calls having a name beginning with "s_" represent calls to external services.

Using similar approaches, several other processing patterns may be easily implemented, and to name only a few:

- Apply Inplace - apply a certain transformation on all the objects,
- Apply New - create a new list based on the transformed objects of a given list,
- Count - count the objects having certain characteristics,

- For Any - check if all objects in the list have certain characteristics,
- Fold - calculate a global value based on the elements of the list.

Another class of execution patterns derives from the basic *ring pattern*. In [66] we have implemented this pattern using three interdependent services. The basic idea of this multiple instances with prior knowledge pattern is that the invoke of the services describing the ring is done in sequence, repetitively, while certain conditions hold. The general structure of the pattern is presented in Listing 4.2.

```
boolean_value = evaluate_condition(condition)
while (boolean_value) {
  value1 = s_first_service(input)
  value2 = s_second_service(value1)
  boolean_value = evaluate_condition(condition)
  input = value2
}
```

Listing 4.2: Basic Ring Pattern

Starting from the examples depicted above, one may imagine an infinite number of combinations. For instance a particularly useful execution pattern, deferred choice, uses several external services to compute the same result over the same object, but using different techniques. Since only one result is needed, the execution ends when any of the calls returns the result. In this pattern, a particular role plays the basic *parallel pattern* which allows starting multiple calls at the same time.

4.3 Composition Technologies and Tools

The requirements that computational systems built to support scientific processes in general cannot be exclusively fulfilled using proprietary technologies such as RMI, CORBA

and RPC. A comparison made by Gray [108] shows that Web Services are usually less efficient in terms of resources consumption than RMI and CORBA. The performance problems introduced by Web Services are not crucial and can be ignored given the benefits that Web Services introduce. If computational overhead introduced by Web Services is significant in the context of a certain application, technologies such as RMI and CORBA should be considered. For the vast majority of applications the impact to application efficiency is small and due to the advantages they introduce Web Services are more and more adopted as the technology for providing services to potential customers.

Both industry and research communities have understood the benefits that automatic composition may provide. Languages that allow a higher level description of the computational steps together with corresponding platforms that automate workflow's execution have several important benefits. On one hand the process of specifying a workflow is more intuitive and less concerned with low level detail such as service invocation mechanisms. On the other hand, automatic workflow management done by a specialized server is more efficient, easier to control and more secure.

Given a set of Web Services that can be used to create a new application, the workflow engine is able to orchestrate these services based on workflow's description. One important problem to solve is how to enable the workflow engine to discover the most suitable Web Service to invoke for the given purpose. One solution is to consider ontologies for describing the Web Service interface and define matching mechanisms. More details of these methods are given in [156, 85]. While this solution is in principle applicable, the diversity of Web Services, their interfaces and data types used makes this approach feasible only for small areas of computation.

Dynamic composition approaches include AI planning mechanisms and ontology based composition. The set of services dynamically selected to solve a particular problem may change from one invocation to another. As a result, a dynamic discovery mechanism must be used at runtime to decide which services should be invoked. The selection of services must meet requirements regarding the functionality and the QoS to be provided.

In this respect, several general problems may appear [169]. The discovery problem, for example, raises two sub-problems that need to be solved at the same time: obtaining a service description and obtaining the location of the service. Reliability constitutes also an issue since services may be occasionally unavailable.

In [156] it is noted that a generally accepted assumption is that each Web service can be specified by its preconditions and effects in the planning context. A specialized language, DAML-S [39] has direct support for AI planning techniques. The state change produced by the execution of a Web service is specified through the precondition and effect properties of the service profile.

As described in [135], the semantic Web vision is to make Web resources accessible by content as well as by keywords. Web services play an important role in this scenario: users and software agents should be able to discover, compose, and invoke content using complex services. The main drawback of this approach is that specifying ontologies may become a very complicated task.

4.3.1 Web Services Orchestration

Web Services rely to a great extent on XML and related technologies for describing their interfaces and the messages exchanged between client and Web Service. Their suitability for automated machine processing has also encouraged the development of specialized languages for describing Web service workflows as XML based languages. The first notable languages to appear, XLANG [158] and WSFL [25], enabled only static composition of Web services [117]. XLANG relies on structured activities, whereas the second one permits the creation of workflows by linking activities.

As demonstrated in [117], the XLANG language is more restrictive than WSFL in the sense that some workflow patterns are not supported. One such example is arbitrary cycles, similar to the 'goto' mechanism used in unstructured programming [192]. The

WSFL was superseded by the BPEL4WS V1.1 standard language which later was enhanced and adopted as a OASIS standard under the naming WS-BPEL 2.0 [37]. Due to its massive support from industry, the WS-BPEL 2.0 called in short BPEL, now represents the industry standard for describing Web Services orchestration. Its pure XML nature that on occasions makes describing workflows difficult motivated IBM to create a hybrid language, BPELJ [134] that allows Java components to be easier integrated with standard BPEL workflows.

Due to the acceptance of BPEL as *de facto* standard for describing Web Services orchestration, researchers have also investigated the suitability of BPEL and related technologies for describing scientific workflows. In order to analyse the suitability and expressiveness of a language multiple perspectives should be used [192, 191]. The language's power comes from the support it offers for existing control flow patterns, data flow patterns and interaction patterns describing the relation between the process and the services it has to interact with. In [198] the author demonstrates the way BPEL is able to support most of the control flow patterns while [44] investigates multiple interaction patterns and the way they can be expressed in BPEL.

Learning from the experience and shortcomings of its predecessors, BPEL tries to provide support for most of the features that industry and research communities found important while trying to keep the language itself simple. BPEL is used to describe composed Web Services as business processes. While the business process itself is seen by an external client just like any other Web Service, the workflow engine that executes the process has the task to interact with partner Web Services that are the actual providers of services. The interaction plan results from the analysis done by the specialist that identifies the Web Services that are needed to solve a particular problem and the control flow and interaction patterns need to achieve its goal. External Web services called by the process are partners playing specific roles in relation with the process.

The full workflow lifecycle identified in [76] is supported by BPEL. Abstract workflows defined in BPEL capture the partners and the control flow of the process while actual

details about the location of Web services to be used can be provided at runtime. Interaction with external Web Services is defined using the port types of the partner services. Data types of the parameters used by partner port types can be automatically discovered from the WSDL documents and BPEL provides XML specific mechanisms to manipulate data based on their specific format. This functionality is required because it is often the case that the output received from one partner must be transformed to the input expected by another partner interface.

Modelling a workflow can be achieved using several language constructs. They are referred as BPEL activities, which are encoded as XML tags in the BPEL document describing the workflow. The most important ones are:

- Communication activities: receive, invoke, reply, pick, on message, on alarm
- Control activities: if, elseif, else, switch, otherwise, while, repeatUntil, flow, wait, exit, sequence, foreach
- Fault handling activities : throw, catch, catchall, terminate, compensate, compensateScope, rethrow
- Data manipulation and scope: assign, copy, scope, validate
- Auxiliary activities: empty, extensionActivity

Some of the activities mentioned represent themselves containers that can hold other activities. For instance, the *sequence* activity instructs the workflow engine to execute one after another the activities that it contains, regardless if they are calls to other services expressed using an *invoke* activity or an arbitrary combination of other activities.

An important requirement raised by scientific workflows is the ability to describe workflows by combining already defined ones. BPEL supports this requirement since a workflow document can be easily integrated into another document. Scopes of existing activities and for the workflow itself can be created to prevent naming clashing for constructs

and variables that are defined to hold data. BPEL also offers support for exception handling and and compensation activities to deal with execution errors that may occur. Results obtained from partner services may be temporarily stored and manipulated using XML based data manipulation mechanisms.

The most important control flow patterns have corresponding support in BPEL through intuitive BPEL activities. The *sequence* activity may be used to describe a sequence pattern, conditional patterns may be expressed *if* and *switch* activities, repetitive patterns may be expressed using *while*, *repeatUntil* and *foreach* activities. The order of execution can be specified on one hand using the structured activities describe above combined with links that may be specified between activities. A link specifies a dependency relationship between a source activity and a target activity. The target activity can only be started if all source activities on which it depends have been successfully completed.

Graphical interfaces, e.g. ActiveBPEL Designer [28], can be used to create abstract or concrete workflows using a visual interface and to assist the user in deploying the resulted workflow. The user has to define the partners that the workflow process must call and to combine these partners using control flow constructs. Once the workflow is specified the user may even test the workflow by using fake partners automatically provided by the ActiveBPEL Designer environment. Such mock partners may be instructed to return a particular value when they are invoked. After the workflow is deployed as a process in the ActiveBPEL workflow engine it can invoked as regular Web Service by an external client. Depending on how the workflow is constructed, each call may create a separate instance of the workflow or it may reuse an existing one.

Any modification in the structure of the workflow requires that a new workflow is deployed and therefore dynamic modifications of its structure are not possible at run time. Workarounds for this issue may still be possible. One solution would be to break the original workflow into multiple smaller workflows and have them executed one after another. Thus, depending on the dynamic status of the workflow it may call one of the existing workflows that can be individually be modified. Another similar option is to

implement workflows that cover most of the possible scenarios and based on the internal state of the workflow and routing plan of the workflow to execute corresponding sub-section of the initial workflow. None of thesestrive solutions is ideal but at least alternatives are available if really needed.

Apart from the solution provided by BPEL language an important effort was conducted in the scope of several research projects with the aim to provide versatile solution for description and execution of workflows. While BPEL's main intent was from the beginning to provide support for Web Services composition, research initiatives have tried to accommodate multiple distributed technologies. This approach is motivated by historical evolution of distributed computing platforms for scientific computations that were developed over time using a wide range of technologies. Existing tools for scientific computations could not be rebuild from scratch and therefore solutions that could still use them had to be found. Since these systems do not target especially Web Services we include them in the category of Grid workflow systems in a broader sense of the Grid term and not restricted WSRF compliant services.

4.3.2 Orchestration in Grid Environments

The number of workflow systems for Grids is quite large and motivated by the interest to provide a flexible way to describe and execute computational steps required by computations specific to science. Although the main middleware solutions for creating Grids, such as Globus, Unicore and gLite provide mechanisms for resource management and discovery, their capabilities for creating workflows are limited. Their intent is to provide solutions for exposing and managing computational resources at a lower level. It is also possible that existing applications are not easy to integrate with Grid middleware products without extensive refactoring. Depending on the nature of the problem to solve, most of the important research communities involved in scientific computing have strived to design and implement tools and frameworks to support their own computational domain. We investigate here some aspects of the main systems for workflow

design and execution, while a more compressive overview can be found in [199].

Triana [111, 130, 71, 177] is a problem solving environment that consists of several layers of components. Triana abstracts task executors as components using Triana custom data types. Once external service providers are wrapped as components they can be used inside the visual tool to build workflows. JACAW [112] may be used to integrate as components any legacy tools implemented in C. Independent computational nodes on which Triana is running are able to advertise their components.

The user can drag components to the worksheet and connect these components using pipes. Workflows can be modelled as DAGs. The lack of cycles restricts the usability of the system since loops cannot be implemented. Web Services and Grid Services can be used within Triana if they are properly wrapped as components following Triana's model. Web Services can also be discovered by querying UDDI registries and automatically wrapped as Triana components. The workflows created using the visual interface can also be exported using a proprietary Triana format or as BPEL4WS workflow document.

Taverna [140] is a workbench for creating and execution of workflows for life sciences. Its main goals is to provide a versatile way to create workflows based on arbitrary services for which no restrictions on data types used are assumed. This gives the important benefit that virtually any service can be used as a executor in a Taverna workflow. On the other hand, enforcing data matching rules and conversion of data from one format to another has to be explicitly described within the system.

Execution units can be easily added to Taverna by querying existing UDDI registries or specific registries implemented by Taverna system. It is also possible to extract service descriptions from other sources such as existing workflows and even Web pages, by using external capabilities of plug-ins or by querying semantic repositories. It is not possible though to dynamically change the address of a service and therefore, static binding is assumed. Workflows may be described as DAGs in which links between

components describe the data flow links between one output port of one processor unit to an input port of another processor unit.

Behind the scenes the SCUFL language is used to describe the actual workflow which is interpreted by the Freefluo enactment engine. To communicate to a certain type of service a processor type has to be defined for SCUFL due to historical reasons. Various processor types are already available for Web Services, local Java programs, services implementing the REST style interface but support for Grid Services is not provided. When defining a workflow, except for the specific links created by the user, any other control flow mechanisms is inferred from the structure of the workflow. To execute a certain task multiple times, the input for the execution unit should be an array rather than a single value. In this case the system will invoke the service, external or locally implemented, once for every data element in the provided array.

Sedna [86, 194] was developed in an early stage as a platform for solving theoretical chemistry problems. They have chosen Globus as middleware for creating and managing computational nodes while BPEL was seen as candidate for orchestration of Grid services created using Globus. A visual workbench allows users to describe workflows using high level components that are stored internally by the application using a high level description language. Execution components that are used to define the workflow in the visual workbench still have to be defined in terms of port types and data types specific to the BPEL and related technologies. One important facility of Sedna is that a workflow can be deployed as a BPEL process to several workflow engines, among the ActiveBPEL.

The actual tasks that are sent to Grid Services are described using the Job Submission Description Language that is supported by Globus WS-GRAM. Based on their investigations, the authors conclude that BPEL and related technologies provide enough support to be considered viable when compared with other similar solutions. Among the most important advantages of BPEL is the support for control flow constructs, and scalability and reliability of existing workflow engines that are heavily endorsed by industry actors.

Load balancing and scheduling tools, such as Condor [179] and PBS [47], are able to manage pools of resources usually available in LAN environments and to effectively use these resources to solve computational tasks. DAGMan, the workflow management component of Condor, is able to control the execution of static workflows expressed as DAGs. Data dependencies among related tasks are specified by referencing data placeholders, e.g files, as input and output dependencies. Based on these dependencies the scheduler may decide if a certain task may be submitted for execution or it has to wait for other tasks to finish. Based on scheduling capabilities of Condor and DAGMan, P-Grade [115] is a portal for creating and managing workflows. Among the types of tasks that can be used to create workflows in P-Grade, executables, MPI and PVM jobs are supported.

CRESS [174, 190, 122, 189] is a tool initially designed for composition of Web Services that was later enhanced to also support static composition of Grid Services. Due to the differences between Web Services and Grid Services, existing workflow engines are not capable of seamlessly interacting with Grid Services. Based on the visual workflow environment that CRESS offers, the user is able to describe worfklows as DAGs in which execution units represent already deployed Grid Services. The workflow is stored internally using a proprietary language suitable for formal verification of the composition. For deployment, CRESS used a translator to a BPEL workflow format that can be deployed in an ActiveBPEL workflow engine.

Globus Toolkit offers the possibility to describe and run remote jobs through its GRAM component. In conjunction with the Globus, Swift [201] can execute workflows specified as input files. Workflow can be described using a functional language, SWIFTScript, which is interpreted by the Swift execution engine. The resulted workflow can be visualized as a precedence graph. It also permits restarting and rerunning workflows with the option to execute only the jobs that were not executed successfully. The Java CoG Kit [121] reunites a set of tools that can be used for expressing and executing Grid workflows. A specific workflow language can be used to describe workflows executed by the Karajan workflow engine. Workflows may contain control flow constructs for creating

sequences of tasks, define tasks that must be executed in parallel, and define execution cycles in a similar way BPEL supports it.

Condor-G [97] is a product that mixes the inter-site communication capabilities of Globus with the job management offered by Condor. As a result, the DAGMan component of Condor-G can be used to describe and execute workflows. For every job the user can see meta-information such as the issuer of a job, the status, the time it was started/ended, the command that started the job. In the case of a failure, DAGMan is able to rerun only the jobs that were not completed successfully. Using a combination of Condor-G and Stork, workflows can be executed over a Grid[74].

Combinations of the tools mentioned above may be possible. This concept is demonstrated in [77]. The Pegasus's main responsibility is to analyse an abstract workflow and to determine an efficient mapping between the tasks to execute to the actual resources that are able to support their execution. Thus, an abstract workflow described in DAX (an XML language for describing DAGs) can be transformed in a concrete workflow. In this case Pegasus provides complementary functionality to Condor which is the actual resource manager and responsible for scheduling tasks' executions.

Symbolic computation services may be part of a computational infrastructure that can be used for solving complex problems. The analysis of the work conducted in the context of building symbolic computing services by projects such as MONET [32], GENSS [137] or MathBroker [43], has led us to the conclusion that dynamic discovery techniques implemented using AI techniques for Web services, in general, and for symbolic services, in particular, are not yet able to provide a wide-scale applicable solution. The discovery process in MONET uses the MSDL ontology language and the MPDL problem description language to retrieve the right mathematical services by interrogating modified UDDI registries. A similar agent based approach is also used in GENSS.

Our approach differs in several respects. First of all, it uses the functionality offered by remotely installed CASs as potential solvers of mathematically described problems. The current system aims to integrate the processing capabilities of the functions implemented

in remote CASs into the within user's CAS system. The discovery process uses as a main criterion of selection the functionality implemented by a certain service to manage a certain OpenMath call object. The OpenMath standard [157] ensures the interoperability between Web services that expose functionality of different CASs.

Previous results obtained in the context of workflow patterns [139] are used within the current approach to provide a higher level of abstraction. Implementation details are hidden and the user can concentrate on the problem to solve and not on low level details of implementation. The user can build arbitrary complex workflows using standard constructs (workflow patterns): the complex symbolic computation process is specified in terms of workflow patterns and not in a specific workflow composition language.

As discussed in the previous chapter, Web Services and Grid Services are the best technologies for exposing CAS functionality. Existing solutions for Grid Workflow management cannot be used without applying extensive modification to the execution platforms or without implementing additional components and adapters. Visual platforms for description of workflows cannot be integrated within the usual environments of CASs. Using such platforms as standalone environments and CAS environments at the same time would make the process of describing and execution of workflows for symbolic computation difficult and error prone. BPEL language is the best choice for orchestrating Web and Grid Services even if it does not provide full support for Grid Services orchestration.

4.4 Composition of CAS Servers Using AGSSO

Existing systems for distributed symbolic computing allow client applications to discover and access remote services but they are not designed to provide support for describing complicated workflows which can be managed automatically by specialized execution engines. Their support for workflow management capabilities is limited and they do not offer specific solutions for storing results or for transferring required data in

a seamless way. While these capabilities may be provided to a certain extent by external components, the computer algebra specialist would have to explicitly integrate them within the algorithms they implement, which is rarely a simple task. Existing frameworks and technologies that are used in other research domains cannot be adopted for symbolic computation systems without tailoring them to specific requirements of symbolic research field.

In Chapter 3 we have introduced a set of components that were designed to support most of the requirements identified earlier regarding usability, interoperability and extensibility. The CAS Server components, the foundations on which our architecture is built were already designed to support interoperability. Each CAS Server provides the same standard set of capabilities including support for discovery, task management and data management, while the data encoding model used relies on already accepted standards. Clients can submit computation requests encoded in one of the accepted formats and retrieve the results based on job identifiers. Re-routing of results to a specific URL identified service is also possible. Task level control capabilities such as pausing, resuming and cancelling tasks is supported at CAS Server level and all obtained results can be stored the CAS Server for later reference and provenance. In order to support discovery, the CAS Servers are able to contact index services and advertise their current state and information about services they provide.

The CAS Server's standard interface makes composition of services they provide easier and more reliable than composition of arbitrary symbolic services. The standardized interfaces provide external client the guarantee that the structure of services provided by the CAS Server components does not change over time, even if the symbolic capabilities provided through services evolve. Standard data encoding used for describing requests and computed results is another characteristic that makes composing these services easier. A common problem related to dynamic composition is the variations in data encodings used by the services involved in the composition. An execution engine that needs to invoke two services in a sequence has to adapt the response received from the first service to the data model that the second service is able to understand. This problem

does not arise if a single data model for encoding data is used.

To provide capabilities for description and execution of symbolic workflows of arbitrary complexity, several modifications have to be made to the original architecture. We add specialized components at server side for managing workflow execution and we enrich client components with additional capabilities that support description of workflows. The resulted architecture is depicted in Fig. 4.4. Within this architecture, the AGSSO component and its specialized subcomponents represent the central manager of the whole composition architecture. Its main role is to receive workflow instances described at the client side, to deploy and to manage their execution by coordinating CAS Servers and to store final results of the computation.

Several features already provided by the CAS Server components can no longer work as expected for tasks that are part of a workflow without corresponding support from the AGGSO components. Since AGSSO is a mediator between the actual clients and services provided by CAS Servers, AGSSO must ensure that task level control actions are still available. When a task is paused as result of the user's request the actual request must be handled by AGSSO and if needed, routed to the CAS Server that executes the particular task. Therefore, similar services that are available at the CAS Server level should also be supported by the AGSSO component through its interface.

Symbolic workflows are described at the client side as workflow templates. High level predefined constructs that match the most important execution patterns represent the building blocks that the user can combine. Blended with native CAS capabilities, these constructs provide an easy and intuitive way to describe complex computations. Since the majority of CASs only provide command line interpreters, most of the workflows should be described as scripts specific to a particular CAS. More advanced visual solutions could also be an alternative for CASs that provide a visual interface. Because high level constructs are used the code describing the workflow only contains calls to CAS implemented functions matching workflow constructs. Through this calls the system can be instructed to build the workflow in the format that will be sent to AGSSO. Unneces-

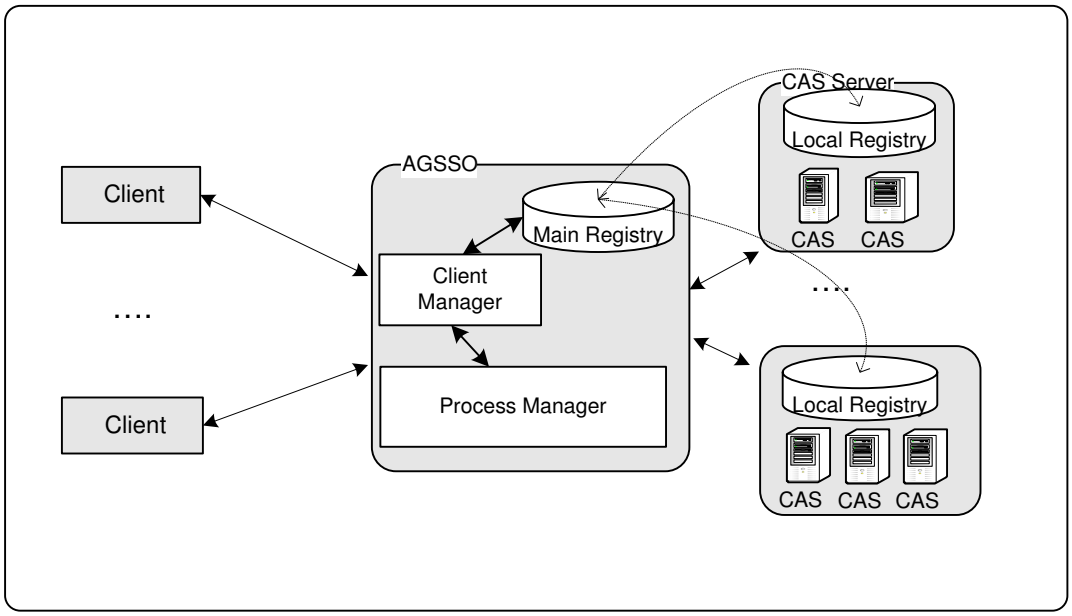


Figure 4.4: Architecture for Grid Symbolic Services Orchestration.

sary details such as actual servers to be used to solve a certain task are omitted at client side because it is AGSSO responsibility to discover the services to use.

The process of specifying workflows is simple and straightforward. As we will further show in Section 5.2, due to its simplicity it can be easily adopted by any CAS. Specific functions available in the CAS environments implement the required functionality to construct the abstract workflow and to wrap and send it along with arguments to an AGSSO component that will manage the workflow further. The XML language used for encoding workflows presented in Table 6.1 is similar to the one of BPEL but it only contains the minimum high level details of the composition. Some details that a complete BPEL workflow contains such as addresses of services to invoke are not required at this level and they will be added later by AGSSO. To demonstrate the viability of this solution we have implemented a GAP specific package. The functions that the package contains do not implement themselves the logic required to construct and submit the abstract workflow. They only represent a thin layer that accesses the functionality provided by a generic component implemented in Java which resides at client side. More details and examples on how workflows are described and particular solutions implemented for

GAP are provided in the next chapter.

For each workflow submitted to AGSSO, the client receives a workflow identifier. The identifier must later be used as a reference for any management task that the client requires, from execution management to results retrieval. Within a workflow, each individual task has its own unique identifier and therefore, functionalities provided by CAS Servers to cancel, pause and resume tasks can still be used by the client by invoking the corresponding operations on the AGGSO interface. The impact that such actions have on tasks and on the workflow as a whole are further analysed in Section 6.2. Depending on the status of the task a request to alter the state of a task will impact the task, a branch of the workflow or even all tasks of the workflow.

The workflow instance received from the client is parsed by the Client Manager subcomponent of the AGSSO and transformed in a template workflow encoded in BPEL. Once the workflow is generated the Client Manager deploys it to the execution engine and starts its execution. During the process of generating the BPEL workflow, required details about which service types are required by the execution are also filled in by the Client Manager. All CAS Servers have the same interface and therefore the invoke logic is the same regardless which is the actual CAS Server that will be selected to be invoked at runtime. The addresses of the CAS Servers to invoke are determined in a dynamic way immediately before a task is submitted to be executed.

Latest information about CAS Servers available, their status and their capabilities are taken from the *Main Registry* subcomponent which is a centralized index. Plug and play components that implement various scheduling strategies can be easily added as subcomponents of AGSSO. These components select suitable services based on functionality mappings between tasks and service providers. A CAS Server is able to handle a certain task if one of the CASs exposed through the CAS Server's interface implements the required operations and OpenMath symbols used to describe the arguments of the targeted operations. The information that Main Registry component provides is guaranteed to be the newest available at the moment when a service is selected because

CAS Servers notify the AGSSO components about any change in the structure of their provided services. A more detailed description of the approach we used to test various scheduling strategies is provided in Chapter 6.

The Client Manager subcomponent has also the role to keep track of the current status of the workflow and of particular tasks. Based on this information it is able to detect which tasks should be planned for execution and which tasks cannot be executed yet due to dependencies to other tasks. When the workflow's execution starts, based on tasks dependencies AGSSO component schedules only the tasks that can be immediately executed. During workflow's execution, whenever a task is solved and the response is received from the CAS Server, the Client Manager analyses if a new task can be started.

In describing a task that is part of the workflow the user has to specify the type of CAS that should be used to compute the task. This information is required because currently it is not possible to determine the most suitable CAS to handle a certain task based on its description. The specified CAS type can be general enough to match a entire class of CASs or it can be refined to target a particular version of a CAS exposed by a certain CAS Server. At the CAS Server side, more than one CASs can be exposed through the same interface but each CAS has a unique name in the scope of the CAS Server even if several machines have the same version of a CAS installed and running. This naming convention makes possible to target a particular CAS installed on a particular machine.

If for instance in the scope of two different CAS Servers two GAP instances are installed, their name is unique and therefore the CAS Server can differentiate among them. For each CAS, the CAS Server provides information about the configuration of the machine it is installed on, e.g. processing power, storage capabilities, but it does not advertise its IP address or machine name. If a user requires that a certain CAS should be handled by a CAS named 'GAP' and 'GAP v3.0' and 'GAP v4.0' are able to treat the request, the CAS Server has the liberty to choose the instance that is more appropriate. On the other hand if more information is added to required type of CAS, such as 'GAP v4.0' and the user especially requests that this particular instance to be used. The same rules apply at

the workflow level when AGSSO component decides which CAS Server to use to solve a task. Even if the whole system is designed to dynamically determine the most suitable CAS Server, and in the context of a CAS Server, the most suitable CAS to execute a task, naming schemes can easily be created to override this default behaviour if needed.

Based on the naming scheme described above even more advanced resource partitioning can be enforced. Let's assume that one group of servers should be exclusively dedicated for long running tasks and for a set of privileged users while other group of servers should be used by general public. This partitioning is easy to achieve only by defining and advertising two separate AGSSO components. The CAS Servers that manage the dedicated resource could advertise their resource to one AGSSO component while the others could be advertised to another one. Anyone that has the right to submit workflows to the first AGSSO component will have their tasks executed on the more powerful servers while the rest of the clients will submit workflows to the another AGSSO component which is only aware of a subset of the CAS Servers available within the architecture.

Bilateral Simple Conversation pattern (Section 3.2) is used by the AGSSO components to submit tasks to a CAS Server and the same pattern is used by the CAS Server to send back the result when the computation has finished. The structure of the request that AGSSO component submits to the CAS Server have not changed. The same conversation pattern is also used for management related requests both in the interaction between the client and AGSSO component on one side and, the AGSSO and CAS Server components on the other side. The client is not aware of the underlying mechanisms and conversation patterns that are used by the the CAS Server and AGSSO. Simple task descriptions provided at client side are transformed to adhere to communication patterns used by CAS Server. Specific initialization steps that need to be run when accessing a Grid Service, data management and security related features are automatically added to the BPEL generated workflow by the Client Manager.

Originally developed for interactions with Web Services, workflow engines do not have native support for interaction with Grid Services implemented using the WSRF specifi-

cation. Moreover, Grid Services use the factory pattern for creating and initializing of Grid Services. It is thus necessary that a client invokes for initialization, first the factory service which creates the corresponding WS-Resource and provides an Endpoint Reference (EPR) to identify the resource that can be used by the client. This interaction pattern is not usually foreseen for regular Web Services and not natively supported. The actual WS-BPEL workflow that the Client Manager generates has also to add required invokes to cope with this requirement.

The actual execution engine that the AGSSO uses for workflow execution is the ActiveBPEL [28] engine. This engine is able to receive workflow descriptions expressed using the WS-BPEL workflow language and to manage their execution. Extensions that would allow the engine to communicate with Grid services and specific hooks for storing management data and results to a local database instance had to be implemented. For dealing with Grid Services, an enhanced version was also provide by [81]. Workflow management capabilities required to control task's execution are not provided by any existing workflow engine. These capabilities are not implemented as additional features of the engine itself. Instead of modifying the engine implementation we use additional hooks and activities in the generated workflow to provide these additional capabilities [64].

The actual size and complexity of the BPEL workflow described in the format required when it is deployed to the ActiveBPEL engine is several times bigger than the abstract workflow generated at client side. Several patterns, such as the sequence pattern, have direct correspondence with existing BPEL activities, but most of the patterns that we provide do not have direct correspondents and therefore they are implemented as combinations of basic constructs. Using the Java API offered by the ActiveBPEL engine we generate constructs similar to those described in [198]. Patterns that can be implemented with minimal efforts are the sequence pattern and the parallel/split pattern because of the direct correspondence for these patterns in BPEL through the sequence and flow BPEL activities. AGSSO is able to support also execution patterns such as conditional and repetitive constructs.

As we mentioned before in Subsection 4.1.1 reusability and reproducibility are important aspects of scientific discovery. To ensure that these requirements are met, the *Main Registry* component stores all relevant details regarding workflows and their execution. A workflow execution request received from a client is stored in the *Main Registry* before any other processing starts. During the execution of the workflow, all details regarding which job is assigned to which CAS Server, the time required for execution and the result obtained are also stored in the *Main Registry*. Each CAS Server also records in its *Local Registry* details regarding the jobs it receives and the hardware profile of the machine that executed the tasks. Combining the information stored in the *Main Registry* and several *Local Registry* components of the CAS Servers executing the jobs, a complete image of the execution can be created.

The AGSSO Server is the software component responsible for execution and management of workflows for symbolic computations. For the actual execution of tasks it selects suitable CAS Servers, submits task to them and collects obtained results. The main features of the AGSSO Server are that it:

- transforms workflows expressed in a generic workflow language to the BPEL format used by the ActiveBPEL workflow execution engine;
- executes scheduling algorithms to select suitable CAS Servers for executing atomic tasks;
- implements a registry that controls CAS Servers to which tasks should be submitted and allows remote users to discover execution capabilities of registered CAS Servers;
- uses capabilities provided by the ActiveBPEL execution engine to execute workflows, monitors and manages workflows' execution;
- through capabilities exposed it allows users to manage deployed workflows which include features for workflow execution management.

4.5 Summary

In this chapter we have addressed the problem of composing functionality of CASs for solving large symbolic problems. The design of the AGSSO Server component that we present in this chapter is covered in [60, 61, 66, 148].

In Section 4.1 we analyse the general requirements raised by scientific workflows and we come to the conclusion that similar requirements are valid for symbolic computation workflows: support for workflows having a large number of tasks and that take a long time to complete; the ability to control and to steer the execution of a workflow; the ability to review and reuse already obtained results; the ability to express workflows in ways that are meaningful for application experts. Most of the current workflow execution engines expect workflows to be expressed in terms of workflow patterns. A short overview of the most important workflow patterns that are often used in symbolic computing is presented in Subsection 4.1.2.

Most of the symbolic problems' solutions are described algorithmically by combining functions implemented by the local CAS instance. By analysing the basic functionality provided by CASs several execution patterns may be identified. In Section 4.2 we demonstrate how basic patterns used in symbolic computations may be expressed using workflow patterns. On one hand we demonstrate that existing workflow engines may be used for handling symbolic computation workflows and on the other hand we provide a set of guidelines to help symbolic computation experts in the process of creating symbolic computation workflows.

Most of the existing workflow execution engines lack support for specific features required by symbolic computations. An overview of tools and technologies used for descriptions and execution of workflows is provided in Section 4.3. The AGSSO Server component described in Section 4.4 reuses capabilities of existing workflow engines and provides additional capabilities to ensure required support for execution and management of symbolic computation workflows.

The actual components executing individual tasks are the CAS Servers. The AGSSO Server receives symbolic computation workflows expressed by clients and transforms them in executable workflow that are deployed to a workflow engine. It provides dynamic selection of computational nodes, i.e. CAS Servers and it provides support for monitoring workflows' execution. Its internal registry retains information about available CAS Server that tasks may be sent for execution and it also stores computed results.

Capabilities already provided by the CAS Server components such as the ability to cancel, pause and resume a certain task are also supported by AGSSO Server. A user can therefore submit and control workflows' execution. The user can experiment different execution scenarios by skipping the actual execution of some tasks of the workflow and manually provide expected result for those task while the workflow is running. To provide this functionality we do not rely on workflow execution engines' capabilities to control the execution of the workflow but on additional hooks that exploit features provided by CAS Servers. This approach is novel because we do not have to alter the behaviour of the execution engine.

The AGSSO component improves the process of describing and execution workflows for symbolic computations. The description process becomes easier and more intuitive because the user only has to describe the solutions in high level building blocks. Such workflow descriptions can be stored for later use, can be disseminated and the solutions may be incrementally combined and improved. Because the actual services to be used are dynamically selected by the AGSSO component, the user does not have to specify such details. The overall efficiency of the system is also improved since components implementing more efficient scheduling algorithms may be added to the system to improve the way available computational resources are used.

Support for reusability and reproducibility is provided by AGSSO architecture by combining information related to workflows' execution stored by the *Main Registry* component of the AGSSO Server and *Local Registry* components of CAS Servers. It is therefore possible to get detailed information about the workflows that were executed,

the machines that were involved in the execution and the time required by individual tasks to be completed. The same workflow can be rerun at a later time and obtained results can be thoroughly documented.

Chapter 5

Generic and Secure Access to Symbolic Services

This chapter describes the design and capabilities of the Client Component of our architecture [60] and the security mechanisms that are used to ensure overall protection of the architecture as a whole and of its components [63]. In Section 5.1 we describe the design of the Client Component. We also discuss the capabilities that this component provides for accessing remote Web and Grid Services. In Section 5.2 we present specific capabilities related to the process of describing workflows for symbolic computations that can be further executed by submitting them to an AGSSO Servers previously presented in Chapter 4. General security mechanisms and the how they are integrated within our components are discussed in Section 5.3.

5.1 Client Component Requirements and Capabilities

Previous Chapters describe the two main components that provide the foundation for building a massively distributed symbolic computations infrastructure. Both CAS Server (Chapter 3) and AGSSO Server (Chapter 4) components rely on Web and Grid Services

for interconnection with other components of the architecture and their capabilities are exposed as Web/Grid Services. Due to the advantages they provide in terms of interoperability and usability, Web Services represent one of the most popular solutions for implementing computational services accessible to remote clients. It is also the case for symbolic computations world in which Web/Grid Services are more and more used for development of new capabilities and even for reimplementing existing ones initially implemented using alternative solutions.

From the client's point of view, Web Services are easier to use that other distributed technologies for several reasons. Software components developed at client side can adopt a wider variety of technologies and platforms as long as they are able to formulate calls specific to Web Services. This is only partially possible with CORBA while RMI requires that Java and RMI specific mechanisms are used to implement both the server and client components. The discovery process is much easier and the description of interfaces is more clear if Web Service are used because the WSDL document provides a description of the interfaces. Additionally, because Web Services rely to a large extent on XML technologies, standardised XML technologies and related tools makes implementing Web Services much easier, reliable and more secure.

Services offered by remote providers represent the building blocks that can be used to build complex computational infrastructures. This architectural model is well suited to the symbolic computations domain and this approach represents an important step ahead in the overall development of computational platforms. On one hand, a large variety of Web Services providing support for solving problems of non symbolic nature already exist and may be accessed by remote clients, including clients specialized for solving symbolic problems that may require support of non symbolic nature. On the other hand a high number of mathematical services that provide support for symbolic and numeric computations already exist and their number is constantly increasing.

Most of the current problems are complex and heterogeneous with regard to specific computational domain. They can only be solved by dividing them in less complex prob-

lems for which a certain solution algorithm may be applied. Therefore it is generally required that more than one computational system has to be used to solve such problems that may involve both symbolic and non-symbolic capabilities.

The NAG library [182] is know as one of the most reliable and complete libraries for numerical computation. Within the framework of the Monet consortium in [128] the authors describe a solution for accessing routines implemented by the NAG library in C programming language as Web Services specific to the Monet platform. Apart from this specific wrapper, Monet provides a framework that allows users to create symbolic services and expose them through a Web Service interface. Similar mechanisms with the ones used in Monet can be used to create new services based on the frameworks provided by MathBroker and GENSS. Additionally, MathBroker project has also investigated the feasibility of exposing services as WSRF compliant Web Services.

Other initiatives such as Maple2G [151] provide solutions for exposing particular CASs through Web Services or Grid Services. A similar solution is the one for Mathematica described in [178]. Even if the components are not interconnected with the use of Web Services they use alternative distributed solutions such as RMI than can be converted to use Web Services with relatively little effort. It is especially the case of systems that use as conversation pattern the Bilateral Simple Conversation because it does not require handling session data at the server side. For the ones such as JavaMath [170] which allows clients to establish computational sessions, the adaptation process is more complicated but it can still be achieved.

Connectors that link various components of our architecture are based on Grid or Web Services and any client that accesses the services provided by our architecture must be able to formulate appropriate calls. Autonomous CAS Server components permit clients to submit requests, to gather results of computations, to manage tasks and to interrogate CAS Servers for provided functionality, all through Grid or Web Service interfaces. Requests sent to the CAS Servers must be correctly formulated not only with regard to the SOAP message but they also have to follow one of the two supported

encoding formats previously presented in Subsection 3.6.1 and Subsection 3.6.2. The AGSSO component uses Web Services to expose its functionality to clients but Grid Services could also be used. Beyond the actual technology used to expose the AGSSO Server interface, the client must also comply with the message format expected by the various operations available. The most complex one is the format of the message request describing new symbolic worflows managed by AGSSO which is further described in Section 5.2.

With the support of the software components installed on the local machine users should be able to access in a seamless way services offered by remote providers. As we stated before, the capabilities to access remote Web or Grid services should be generic enough to support access to generic services and not only to services with a specific interface or that provide a specific type of functionality. These capabilities must be provided within the CASs that computer algebra specialists use and not as additional software tools that force the users to switch to a different environment. From the usability point of view, it is also more convenient to access services directly within the CAS's interface because obtained results may be required for further processing within the CAS environment. We have therefore designed the client components of our architecture to be easily integrated within existing CAS environments. With the aid of several generic add-on components CASs may be enriched to provide access to both arbitrary Web or Grid Services and to specific services provided by the server components of the AGSSO architecture.

Usually the user only knows the address of a remote service or even the address of a UDDI registry or a Globus Grid Services container. To invoke a certain service the client must be able to discover and select the service it wants to access. Moreover, the client component that prepares the calls has to be aware of the service's interface. Access to generic services can only be achieved if support is provided for the entire process of interacting with remote services, from discovery of services and their interfaces to the actual call and result parsing. Since services are not known a priori, dynamic clients that are able to interact with external services must be generated. A wide range of parameters must be considered when generating Web service clients automatically. These include

the service address, the port that the service provider listens to, the number and names of the methods or the way that the service descriptor is obtained.

Several tools provide partial implementations of Web service automatic client component generators. Eclipse [4] can generate a Java Bean proxy from a WSDL document for Web services deployed on WebSphere servers. Webservice Studio [23] can be used to invoke Web service operations interactively within a testing environment for services of which endpoint is known. It fetches the WSDL and based on service's description it generates a proxy from the WSDL and displays the list of methods available. The user can then choose any method and provide the required input parameters. Systinet Developer for Eclipse [20] also supports client generation, the entry point being the WSDL document describing the Web service and automatically generates Java client code that calls the Web service; the developer must create the real method calls on the prepared interfaces in the client code. Other solutions to generate the Java classes needed to invoke a Web Service programmatically are Novell exteNd Workbench [181], JAX-RPC Stylus Studio [185], etc.

The ASSIST [35] framework aids the application developer by providing them with a proxy library whose entries are the stub methods for the remote Web service. These are generated from the services WSDL file. The programmer must instantiate the stubs with the code needed to invoke the services methods, and place calls to the stub methods within the code provided by the framework modules. A different approach is taken by Xydra-OntoBrew [99]. This provides on-the-fly WSDL to Web-form generation for simple services and portlet clients: the Xydra servlet takes WSDL as input, generates an XHTML form that allows the user to provide an input message, gathers the submitted input values and converts name-value pairs into an XML message that is sent to the Web service. Finally, it displays any result messages. While Xydra is a sophisticated response to the client code creation problem, simpler solutions are needed especially when workflow execution of combined services is desired.

The tools mentioned above have some limitations and inconveniences related to the pro-

cess that needs to be followed when implementing clients for remote services. They also cannot be easily integrated with existing CASs because the process of code generation is only partially automated. Additionally, these tools do not include Grid service capabilities, and, as far as we are aware, there is no previous automatic generator that supports WSRF-compliant services. MathGridLink [178] is a solution specific to Mathematica for deploying Grid Services and accessing these services. The Maple2g [149] provides a mechanism that can be used in Maple to submit and obtain result of Grid tasks. Its capabilities are build over the standard Globus GRAM service. Therefore the tools mentioned above are not able to support access to generic Grid services but to services having a particular interface and usage pattern.

To a certain extent the CAS itself has to implement features required to support the process of interacting with remote services, one of the most important requirement being to handle data encoding in specific encoding formats. Remote service implementing symbolic computation capabilities may expect that request parameters are encoded using a specific encoding format such as OpenMath. CAS Servers use OpenMath and SCSCP compliant messages but other services may require other encoding models. Since the CAS is the direct beneficiary of the services, it is its direct responsibility to ensure that it is able to encode and decode specific data formats that are out of the scope of Web Services in general. Except capabilities that are specific to symbolic computations, the rest of the capabilities that are related to accessing remote services should be provided by external components that can be accessed using native CAS routines. External packages such as Apache Axis [40] provide such capabilities in a reliable way and they can be integrated with existing CASs through adapters.

Within this chapter we describe a client side suite of packages that provide an easy to use solution that can be used in conjunction with virtually any CAS. As a result, CASs can be enabled to access external Web and Grid Services. Consistent security mechanisms must be enforced within our AGGSS architecture to ensure that both server side components and clients are effectively protected against security threats specific to distributed environments. The second section of this chapter provides an overview of

security mechanisms available in the context of Grids and describes how they are applied within AGSSO.

5.1.1 Enabling CASs to Access to Grid and Web Services

The architecture of the client side component is composed of two types of components, the ones that have the role to provide capabilities for interacting with generic remote Web or Grid Services and a special category that are designed to be used in conjunction with the services provided by the AGSSO components. All these components rely themselves on specialized components and packages developed by third party providers. For instance, components at the client side that allow access to Web Services rely for some of their implemented features on the capabilities offered by Apache Axis. Similarly, other components provides support for features specific to Grid Services and rely on already existing APIs provided by Globus Toolkit. Whenever possible existing solutions already well known and accepted by research communities and industry are preferred to ad-hoc solutions. The client side components provided by AGSSO have the role to embed features such as security certificates management and further provide them as routines accessible directly within the CASs' command line environment. Due to specific design of the components we developed, they can be easily integrated within any CAS.

In addition to the core functionality that allows access to remote services specific functionality for managing data and preparing calls is provided. We have presented in the previous chapter a solution for handling symbolic computation workflows. This solution required that the client specifies the workflow in a specific format which can be understood and managed by the AGSSO server. Even if the workflow instance that has to be described at client side does not have a complicated structure, the fact that it must be described using the specific XML language makes this task cumbersome to be done manually. One of the components available at client level implements functions that assist the user in creating workflows in the appropriate format. All features mentioned

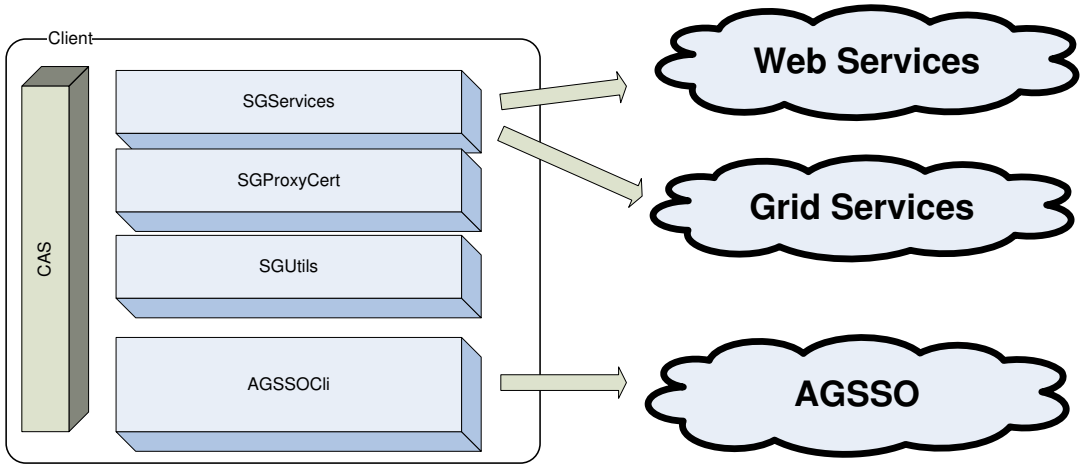


Figure 5.1: Client Side Architecture

above are wrapped into a single stand-alone executable that can be accessed by any external component which is able to communicate through pipes. This represents the first, the lowest level layer of the client side architecture.

The second layer of components that must exist at client side are components that are directly integrated within the CAS and they have the role to facilitate access within the CAS environment to the features provided by the first layer. Within the CAS's development environment, the user should access already provided routines that makes accessing Web Services or describing complicated workflows simple and intuitive. Packages of functions specific to a certain CAS represent a thin layer and have the role to relay requests to the specialised external components mentioned above. This decoupling is in many respects beneficial. Specific functionality does not have to be implemented within the CAS and only a thin layer of routines which formulate appropriate requests to the first layer have to be provided. Enabling a new type of CAS to access Web of Grid services is easy and reliable because the core functionality is provided by external components. If access to new technologies have to be provided within the CAS, they can be easily added at a later time.

The general architecture of the client is presented in Figure 5.1. CASs access the func-

tionality of the CAGS components by communicating with the RunManager component which is a command line interpreter. RunManager is a completely generic interpreter that exploits Java reflection capabilities to allow the execution of any class. The sub-components of the clients side helper component are the following:

- *SGServices*: this provides support for three types of operations retrieval of the list of services registered to a certain UDDI registry or Globus container; retrieval of signatures for the exposed operations of a service; calling remote operations.
- *SGProxyCert*: this handles issues arising from the need to support single sign-on for users of the Grid and delegation of credentials: namely the creation and destruction of proxy certificates. The component can also be used to retrieve information about the owner of a X509 certificate and about the lifetime of a proxy certificate. Since a user may have more than one X509 certificate, but with only one being used at a certain moment, when creating a proxy certificate, the location of the certificate is automatically stored in a session file. When a proxy certificate is needed, this default certificate is loaded and used.
- *SGUtils*: this provides additional functionality for explicit file transfer, file deletion and remote job execution. This capabilities are related to generic services that Grid environments build over Globus provide. For data specific management that the AGSSO architecture provides, other services are involved which the user does not have to call explicitly.
- *AGSSOCli*: this package reunites all the functionality required for users to access capabilities provided by the components of the of AGSSO architecture. This component is responsible for incrementally constructing the workflow in the format that the AGSSO server is able to understand and also specific capabilities to interact with services provided by AGSSO and CAS Server components

5.1.2 Use Case Scenario to Access Generic Web/Grid Services

One of the most important capabilities provided by CAGS is access to remote Web and Grid Services. Part of the discovery process several capabilities are available: obtaining a list of Grid/Web services registered at a certain URL; obtaining the signatures of Web and Grid Services operations; calling an operation and retrieving the result of an operation call. Secondary functionality related to arbitrary services include file transfer using specialised Grid Services, job submission through WS-GRAM services, and management utilities for handling X.509 proxy certificates. A typical scenario of accessing a Web or Grid Service usually has as a first step the discovery of a service by consulting a service's details based on a registry URL (either a UDDI registry or a Globus Container). Listing 5.1 depicts a typical execution scenario at client side.

```
1.    start scenario(registry_URL)
2.        if (is_Web_service_registry(registry_URL))
3.            service_list:= get_Web_service_list( registry_URL,
                                                   toMatch,
                                                   options)
4.        else
5.            service_list:= get_Grid_service_list( registry_URL,
                                                      toMatch)
6.        endif
7.        service:= select_service(service_list)
8.        operation_list:= get_operation_list(service,toMatch)
9.        operation:=select_operation(operation_list)
10        [create_proxy_certificate();]
11.       result:= call_operation(service,operation, parameters)
12.   end scenario
```

Listing 5.1: Access to Web/Grid Services

Here, *registry_URL* parameter is a valid URL of a UDDI registry or a Globus container. The toMatch parameter is a selection string that must be a substring of the service name in the get Web service list/get Grid service list combined with a substring of the opera-

tion name in the get operation list. The selection functions *select service/select operation* are user-defined functions that can be used to select the desired service/operation. Note that this scenario assumes that the user only knows the registry URL. If the user already knows, for instance, the service URL and the signature of the operation, then the unnecessary steps can be omitted.

5.1.3 Summary

Computer algebra specialists must be provided with simple and intuitive features within the environment of the CAS they use to allow them to access remote Web and Grid Services. They have to be able to access generic services but they also have to be provided with additional support for describing and deploying workflows for symbolic computation.

The Client Component of our architecture provides:

- Features to discover and call remote Web and Grid Services;
- Features to support standard security mechanisms required by Grids, including management of security certificates and establishing secure connections;
- Features to support description and execution of workflows. Workflows described by the user are further submitted for execution to AGSSO Servers;
- Features to support workflow management, retrieval of results, steering of execution.

5.2 Workflow Description

Another important capability that is provided within CASs' development environment enables users to describe and deploy symbolic computation workflows. Once submitted to an AGSSO Server, the workflow is parsed and executed according to the scenario

previously presented in Section 4.4. Application specialists have to be provided with constructs that allow them to describe workflows in a seamless way. Some of the details required for the workflow to be complete and compatible with the workflow engine used at AGSSO component level can be automatically added at server side and therefore they can be skipped at client side. The user should only describe the solution for a certain problem in terms of workflow patterns and data links between atomic tasks of the workflow. In Subsection 4.1.2 we have described the most common patterns that we have identified in symbolic computations for which appropriate constructs should be available. In the following table we describe the mapping between the workflows and the functions available within the GAP environment that can be used to construct workflow patterns. These functions are grouped under the package SWIP. Similar solutions may be easily provided for other CASs with a minimal effort.

Several components at the client side are involved in the description of workflows. Specific packages implemented at CAS level have to provide functions that map the workflow constructs. Each call to such functions is forwarded to be handled by the AGSSSO-Cli client support component which is the actual component where the workflow is constructed in the format in which it will be sent to the AGSSO server. Once the workflow is created the user may submit it for execution to a prior known AGSSO server address. Supported constructs for workflow description include constructs for sequence patterns, conditional patterns, repetitive patterns and declaration of variables that may be used as data containers. Variables declared at workflow level are of special nature because they are meant to be used at server level at workflow run time as opposed to variables that are used at CAS level which are only valid in the context of the CAS. To demonstrate these capabilities and the way they can be integrated within a CAS environment we have implemented a special package for GAP.

Variables of the workflow are automatically managed with the direct support of the workflow execution engine. The user may declare such variables and may even assign initial values to these variables. Variables will be used to store intermediate values during the execution of the workflow and therefore their state will only be modified as a result of

XML WORKFLOW TAGS	GENERATING CAS FUNCTION
<workflow>	SWIP_startWorkflow()
</workflow>	SWIP_endWorkflow()
<sequence>	SWIP_startSequence()
</sequence>	SWIP_endSequence()
<multichoice>	SWIP_startMultiChoice()
</multichoice>	SWIP_endMultiChoice()
<branch>	SWIP_startChoiceBranch(condition)
</branch>	SWIP_endChoiceBranch()
<if>	SWIP_if(condition)
</if>	SWIP_endIf()
<trueBranch>	SWIP_if(condition)
</trueBranch>	SWIP_else()
<falseBranch>	SWIP_else()
</falseBranch>	SWIP_endIf()
<parallel>	SWIP_startParallel()
</parallel>	SWIP_endParallel()
<foreach>	SWIP_startForeach(initValue, endValue)
</foreach>	SWIP_endForeach()
<while>	SWIP_startWhile(condition)
</while>	SWIP_endWhile()
<invoke .../>	SWIP_invoke(CasID, command, varReference)
<variable name=var1> $value </variable>	SWIP_declareVariable(varName, varValue)
<casID>	SWIP_invoke(CasID, command, varReference)
<call>	SWIP_invoke(CasID, command, varReference)
<initValue>	SWIP_startForeach(initValue, endValue)
<endValue>	SWIP_startForeach(initValue, endValue)

Table 5.1: Mapping between XML workflow language and GAP functions.

workflow's execution steps. At CAS level references to these variables may be used in two main situations:

- For building conditional expressions that appear as part of conditional patterns
- To mark data dependencies between different tasks

Variables that are meant to be used in expressions that are part of conditional or repetitive patterns, e.g. *while* or *if* constructs, may only contain numerical values encoded as plain

strings. These numerical values may be used to express conditions and their value can be be modified by storing results of remote service invocations when they are provided as the third parameter of the SWIP_invoke function. This type of variable must be declared using the SWIP_declareVariable function. A typical declaration of this type of variable is:

```
v1 :=SWIP_declareVariable("1");
```

The call of the SWIP_declareVariable function has as effect the registration a new variable at workflow level to which the value of "1" is assigned. The local variable *v1* is initialized with a value that represents the name of the variable in the context of the workflow, e.g $variable_1. Whenever *v1* is passed as argument of one of the functions of the SWIP package, it will instruct the AGSSOCli component to record the appropriate use of the workflow level variable.

Valid conditions are expressions that can be evaluated to boolean values by a subset of rules defined by the XPath standard. This subset is for the moment limited to composing simple expressions that contain decimal numbers, boolean and comparison operators, grouping parentheses and variable handlers of workflow declared variables. Examples of valid conditional expressions include:

```
5 < 4
1 >= $variable_1
```

where $variable_1 represents a workflow variable that contains a value that may be evaluated to an integer when parsed by the workflow engine following internal variable evaluation rules.

The second category of variables contains variables that are used to link two or more invoke activities with data pipes. If one invoke activity should use as input the result that was produced by calling other services, these dependencies are expressed using the second type of variables. During the workflow's execution these variables contain values encoded in OpenMath and they cannot be used directly to build conditional expressions.

A call to the SWIP_invoke() function of the SWIP package will return at CAS level the name of a workflow level variable in which the result of the invoke will be stored. This variable can afterwards be used as input for other invokes. The call at CAS level:

```
aVar:= invoke(...);
```

specifies that the CAS variable *aVar* stores a handle to the result obtained through invoke. This handler can be used in a following call to mark that the result is used as input data for another invoke. This is possible because the general format of a call is:

```
aVar:= invoke('CASID', call)
```

where *call* is a string that either describes a function and parameters or an OpenMath object in the SCSCP format. Let aVar1 and aVar2 be two local variables that hold handles $variable_1 and $variable_2 referencing results obtained by previous invokes. To obtain the call of a remote function the call

```
aFunction($variable_1, $variable_2)
```

may be obtained by concatenation:

```
Concatenation("aFunction(",aVar1,",", aVar2,")";
```

5.2.1 Workflow Examples

In order to demonstrate the way different constructs may be used within GAP we provide several simple examples that have the role of clarifying how the functions of the SWIP package may be used to describe workflows. The example in Listing 5.2 links in sequence two functions that calculate the factorial for a given integer value. The first call calculates at line 6 *factorial(3)* while the second call at line 9 uses as input the value obtained as output from the first call. Before issuing the actual invokes, the values that have

to be passed as parameters are constructed as OpenMath objects by using GAP string manipulation capabilities, e.g. a call to the Concatenation function and specific functions for constructing the OpenMath representations, e.g. the SWIP_getSCSCPFormat() function.

```
1.  LoadPackage("swip");
2.  SWIP_startWorkflow();
3.    SWIP_startSequence();
4.          i1 := SWIP_getSCSCPFormat(
5.                          "scscp_transient_1.WS_factorial(3)");
6.          v1 := SWIP_invoke("GAP",i1,"");
7.          i2 := SWIP_getSCSCPFormat(Concatenation(
8.                   "scscp_transient_1.WS_factorial(",v1,")"));
9.          v2 := SWIP_invoke("GAP",i2,"");
10.     SWIP_endSequence();
11. SWIP_endWorkflow();
```

Listing 5.2: Sequence in GAP

The second example presented in Listing 5.3 depicts how two independent calls may be run in parallel. The SWIP_startParallel() and SWIP_endParallel() have the role to mark the invokes that should be started in parallel, in our case the invokes at lines 6 and 9.

```
1.  LoadPackage("swip");
2.  SWIP_startWorkflow();
3.    SWIP_startParallel();
4.          i1 := SWIP_getSCSCPFormat(
5.                          "scscp_transient_1.WS_factorial(3)");
6.          v1 := SWIP_invoke("GAP",i1,"");
7.          i2 := SWIP_getSCSCPFormat(
8.                          "scscp_transient_1.WS_factorial(6)"));
9.          v2 := SWIP_invoke("GAP",i2,"");
10.     SWIP_endParallel();
11. SWIP_endWorkflow();
```

Listing 5.3: Parallel Execution in GAP

Execution of a certain action for a number of times is a frequent requirement especially when processing of a list of objects is needed. In Listing 5.4 we assume that we know beforehand the total number of executions required and we only need to invoke one or more service for the given number of times. Since the worfkow's execution is done at server side, the variable that counts the number of service invokes done at a certain stage of execution has also be managed within the workflow. The solution is to use a variable that counts the number of invokes and update its value each time an invoke was completed. Therefore, at line 3 we declare a new workflow variable and we assign to it the value of 1. At line 4 we specify that the content of the while loop should be executed if the condition holds. In our care the value of the workflow variable is smaller that value of 5. The example demonstrates how the value of the worflow variable should be updated, namely by invoking an external incremental service at line 7.

```
1.  LoadPackage("swip");
2.  SWIP_startWorkflow();
3.      v1 :=SWIP_declareVariable("1");
4.      SWIP_startWhile(Concatenation("5␣>␣",v1));
5.          i1 := SWIP_getSCSCPFormat(Concatenation(
6.                "scscp_transient_1.WS_increment(",v1,")");
7.          v1 := SWIP_invoke("GAP",i1,"");
8.      SWIP_endWhile();
9.  SWIP_endWorkflow();
```

Listing 5.4: Repetitive Pattern in GAP

In Listing 5.5 we present the implementation of the ring workflow. For improved readability we will omit to explicitly show how variables and values used in the GAP client where encoded to be sent to the workflow engine. For example, in line 4. we write:

```
SWIP_startWhile("$n < 10");
```

while the correct GAP syntax should be:

```
SWIP_startWhile(Concatenation(n,"<10"));
```

because *n* is defined as a local variable in GAP that holds the reference to a variable declared within the workflow. In the example presented in Listing 5.5 we create a ring of three services. One service invoked in line 6. is responsible for incrementing the value of the variable *n* and storing the result in the same variable. In line 7. the value held by *n* is sent to another GAP instance to evaluate if the number is prime or not. The result of the evaluation is stored in the variable *m*. After the call the value held by *m* will be "0" if the number is not prime and "1" if the number is prime. Based on the value of *m* the value held by *n* is stored or not in line 10.

```
1.     SWIP_startWorkflow();
2.         n := SWIP_declareVariable("0");
3           m := SWIP_declareVariable("0");
4.         SWIP_startWhile("$n␣<␣10");
5.             SWIP_startSequence();
6.                 SWIP_invoke("GAP1", "Inc($n)", "$n");
7                  SWIP_invoke("GAP2", "IsPrime($n)", "$m");
8.                 SWIP_startMultiChoice();
9.                     SWIP_startChoiceBranch("$m␣==␣1");
10.                         SWIP_invoke("Maple", "Store($n)");
11.                     SWIP_endChoiceBranch();
12.                    SWIP_endMultiChoice();
13.            SWIP_endSequence();
14.        SWIP_endWhile();
15.    SWIP_endWorkflow();
```

Listing 5.5: Ring Pattern in GAP

The "ring workflow" example demonstrates a slightly more complicated workflow in terms of structure. A similar workflow can be used to implement a solution for the orbit enumeration algorithm [126].

5.2.2 Workflow Level Task Management

One of the features that are important for controlling the behaviour of workflows is execution management of submitted tasks. The execution of a workflow is automatically managed at the server side and the user does not have to issue a specific command for the workflow to be started. Once submitted, parsing of the workflow instance and execution of individual tasks is done as soon as the required resources are available and the internal state of the system permits it. Since the execution of certain tasks of the workflow may take a long time to complete it is important to enable the user to control workflow's execution by issuing appropriate calls to the AGSSO server were the workflow was sent for execution the user can request pausing or resuming of a workflow or of individual tasks. Moreover, it is possible to change the result for a certain computation by manually assigning a value to a certain task that overrides the actual result of the computation for that task.

Functionality provided by the AGSSOCli component allows the user to retrieve information about the structure of a certain workflow already submitted for execution and the individual status of tasks based on a workflow or task identifier. This functionality lets the user to verify that the submitted workflow was correctly parsed by the system. Additional to the workflow's structure the user receives information about task identifiers of each task of the workflow and execution state for each task. Using the workflow and specific task identifiers the user may alter the normal execution of the workflow.

Issuing a pause command can be done both at workflow and task level. If a task level pause is issued, the task and all tasks that depend on the paused task are affected. The pause request propagates to all tasks that have a dependency relation to the paused task. In a similar way, resume of a task has also an impact not only on the resumed task but also on the dependent tasks. Cancelling of a workflow is also possible. Issuing a cancel command has as result immediate cancellation of all tasks and freeing all resources used by the workflow. Since workflow managers are not able to provide this functionality through built-in features, additional hooks implemented at workflow and database level

have to be created to provide this behaviour.

5.2.3 Summary

With the support of the Client Component integrated within the CAS, computer algebra specialists may describe workflows by combining workflow execution patterns as building blocks: sequence, parallel, conditional and repetitive. For an already submitted workflow, the user may obtain the status of the workflow or of a certain task part of the workflow. Based on task identifiers the user can pause and resume tasks, can cancel tasks, manually set result values and inspect already computed results.

5.3 Security for Symbolic Services

In Subsection 5.3.1 we briefly discuss the most important concepts related to security of Grids. In Subsection 5.3.2 we discuss security mechanism used to ensure the security of our architecture and the support provided for interacting with third party secure services.

5.3.1 Common Security Standards in Grids

Wide adoption of Grid technologies to build systems for scientific computing purposes introduces the need of a careful consideration of the possible security threats that systems based on these technologies are exposed to. Grid infrastructures are usually built upon public communication infrastructure therefore they are vulnerable to common attacks for the Internet world. The main types of security issues that Grid has to consider are architecture related, infrastructure related and management related [68]. Confidentiality of the data shared in the Grid environment and mechanisms to ensure authentication, authorization mechanisms for access to resources and quality of service related issues are part of the first category. The second category comprises security problems related

to host components and secure transport of data over the wire. The third one refers to problems related to credential management, trust and monitoring services.

Trust among the participants that share and access remote resources is a central concept of security and it is vital for Grid infrastructures. Virtual Organizations(VO) are established based on the willingness of resource providers to share their computing capabilities to other members of the VO. Reliable authentication mechanisms must therefore be enforced to ensure that users and resource providers cooperate in a secure environment. The Grid Security Infrastructure (GSI) provided by Globus addresses these issues related to security and provides solutions for secure communication. Standardized security mechanisms enforced at Grid level ensure that security breaches are less frequent, easier to detect and address.

The communication among various partners of a VO built using Globus middleware relies to a great extent to HTTP and HTTPS communication protocols. Additional protocols such as GridFTP may also be used for data transfer. The secure communication mechanisms implemented by Globus GSI implements secure communication by targeting two communication levels: transport and message. Transport level security applies TLS encryption for all communication that is sent over the wire. This type of encryption guaranties confidentiality and authentication at least from the server side and optionally client authentication may also be enforced. Integrity of the messages exchanged is also ensured through transport level encryption. Message level encryption may be used to encode only the message content while the rest of the communication is transmitted as plain text and therefore confidentiality, authentication and integrity of the messages is ensured. If authentication of the client and integrity of sent messages are required while confidentiality is not mandatory a slightly more efficient solution is to send messages in plain text and to append digital signatures that guarantee the authenticity of messages and the identity of their originator. All these features are based on X.509 standard.

Currently the most popular and wide spread solution for users' authentication in Grid environments are solutions base on X.509 Public Key Infrastructure (PKI) [29], but other

standards such as Kerberos Network Authentication Services [138] or plain security credentials management are also used. Anonymous authentication is possible especially when the client's identity is not particularly relevant. Globus GSI provides native support for authentication based on X.509 certificates but additional solutions that allow integration with Kerberos based systems are available. Since authentication of the actors participating in a distributed architecture can be achieved with different authentication protocols, e.g. Kerberos and PKI, an interoperability between such services may be sometimes required. A service responsible for providing a X.509 certificate based on a Kerberos authentication was previously reported in [30].

VOs can easily be built over a hierarchy of Certificate Authorities that guarantee authenticity of X.509 certificates exchanged amoung participants in the VO. The use of security certificates provides further functional benefits. Single sign-on, delegation of credentials and with them finer grained control over the identity of the users that are entitled to access specific hardware and software resources can be easier provided. Delegation of credentials is particularly important mechanism if proxy components have to access secure services on behalf of a client. The proxy itself may not be entitled to access particular resources but while it acts on behalf of the client it may use its certificate to access them. To support credential delegation Globus provides the Credential Management Service and the MyProxy component [45].

For Grid authentication purposes, each user has an X.509 certificate. This certificate could be stored on the user's system, but this solution makes them vulnerable to theft by trojans or viruses. MyProxy acts as an on-line credential repository for X.509 security credentials which are composed of private keys and certificate tuple. Based on stored credentials MyProxy generates proxy certificates, all via a TLS secured network connections. Since proxy certificates expire after a relatively short period of time, usually 12 hours, the user must periodically renew them. The generated proxy is then stored in the repository and is accessible via (username, password) combination. The password is chosen by the user when first generates the proxy and has the same lifetime as the proxy certificate. Additionally, MyProxy can be accessed through the network from various

locations which proves to be an important advantage for mobile clients.

GSI offers implementations for both WS-Security and WS-SecureConversation. With the latter, a security context is established resulting better performance when multiple invocations take place among two communication partners. Credential delegation is only available if a security context is established through WS-SecureConversation. Authorization mechanisms implemented by Globus GSI can be applied at Globus container level, at service level and at resource level. The authorization scheme can be implemented through authorization descriptors or dynamic Java settings. The default authorization mechanisms can be extended by custom authorization handlers. An example of an extended authorization mechanism is provided by POSITIF [72].

5.3.2 Security for SymGrid-Services Architecture

Security plays an important role for the SymGrid-Services architecture. To ensure that security is properly enforced within the architecture two aspects have to be considered. On one hand, the components that are part of the AGSSO architecture have to provide appropriate security features as part of their standard capabilities. On the other hand for components that are designed by third party providers such as independent services that do not follow our design constraints, appropriate procedures and protocols that ensure secure access to them have to be provided if components of the AGSSO architecture behave like clients to such components. Third party services may themselves require that certain security protocols are implemented by their clients. We also need to prevent to the highest extent possible security problems that may arise by invoking malicious services.

The main component types that AGSSO is composed of are CAS Servers responsible for exposing CASs functionality through Grid Services interfaces, AGSSO components responsible for managing workflows, and client components that formulate requests. Additionally, the client components, through the CAGS module described in Section

5.1 support users for accessing generic service providers that do not necessarily follow the architectural constraints imposed by AGSSO. Appropriate support for security has therefore to ensure that services themselves are protected on one hand and on the other hand that client components are able to access provided services in a secure way.

Secure communication can only be established if we consider and eliminate the security threats specific for communication protocols that are used by the Grid middleware. Since the architecture is built upon the public architecture of the Internet,security issues such as the integrity, confidentiality, authorization and QoS requirements must be validated for the use cases that the system is supposed to support.

The main types of users that will interact with the system are regular users and system administrators. Regular users access the system through software components that are part of their CAS of choice environment. It is also possible that more advanced users implement themselves functionality to access remote services that do not necessarily rely on existing CAGS components provided as part of AGSSO architecture. Additionally, regular users may affect the security of the system by implementing CAS level functions that are exposed later as services it they are not properly evaluated by the administrators of the system. Administrators are privileged users that are able to control the way various components of the system behave. Each CAS Server and each AGSSO server may have their own administrators and part of their responsibility is to assess whether particular functions exposed by CASs should be exposed as Grid Services. More advanced configurations may also be possible if VO level authentication is used and specific administrators gain the privilege to control more than one CAS Server or AGSSO component. Even if they are part in a bigger computational infrastructure, each component is still autonomous and should be possible to control their configuration in an independent way.

Client Side Security Features

Security mechanisms at client side must ensure the integrity of data transmitted over the wire and whenever possible authentication of the services that the client is interacting

with. Authentication of the remote service is of high importance for client's security and its trust in the partner service. These are fundamental in the context of credential delegation which will further allow the services to contact other services using client's credentials. Results obtained following a request for a computation can also be trusted and considered as correct if the client is confident in the actual identity of the remote service. The client has to identify itself in relation to the remote service if the security policy of the service requires so. While in some cases username/password based authentication may be available, enforcing authentication mechanisms such as those provided by X.509 certificates represent standard security approach for Grid Services.

The AGSSO infrastructure is available at client side through two subcomponents of the AGSSOCli components, one specific for accessing services provided by AGSSO servers and one that helps the user in the process of defining symbolic computations workflows. To submit workflows to an AGSSO server component that has a security mechanism enabled, several security steps must be executed. Using the MyProxy certificate repository requires mutual authentication between the client and AGSSO server. Before the user starts using AGSSO, the user stores his credentials in the MyProxy repository. At subsequent calls the user must only provide his user name and password which are required to enable the AGSSO server to obtain a proxy certificate. The AGSSO server uses the proxy to further access resources on behalf of the client (see Figure 5.2). All communication between the client and MyProxy server is achieved through a private (encrypted) TLS channel. The same security measure is applied for communication between the Computer Algebra System or Portal and the ClientManager component of the AGSSO server.

The steps required to access services provided by AGSSO are consistent and straightforward, therefore easy to follow. For accessing arbitrary services though, the client component has to be versatile enough to cope with various security mechanisms implemented by third party services. The client side CAGS component provides support for dealing with security features that enable the client to access third party Web and Grid services. Unfortunately, for plain Web Services, issues such as trust and QoS do not have

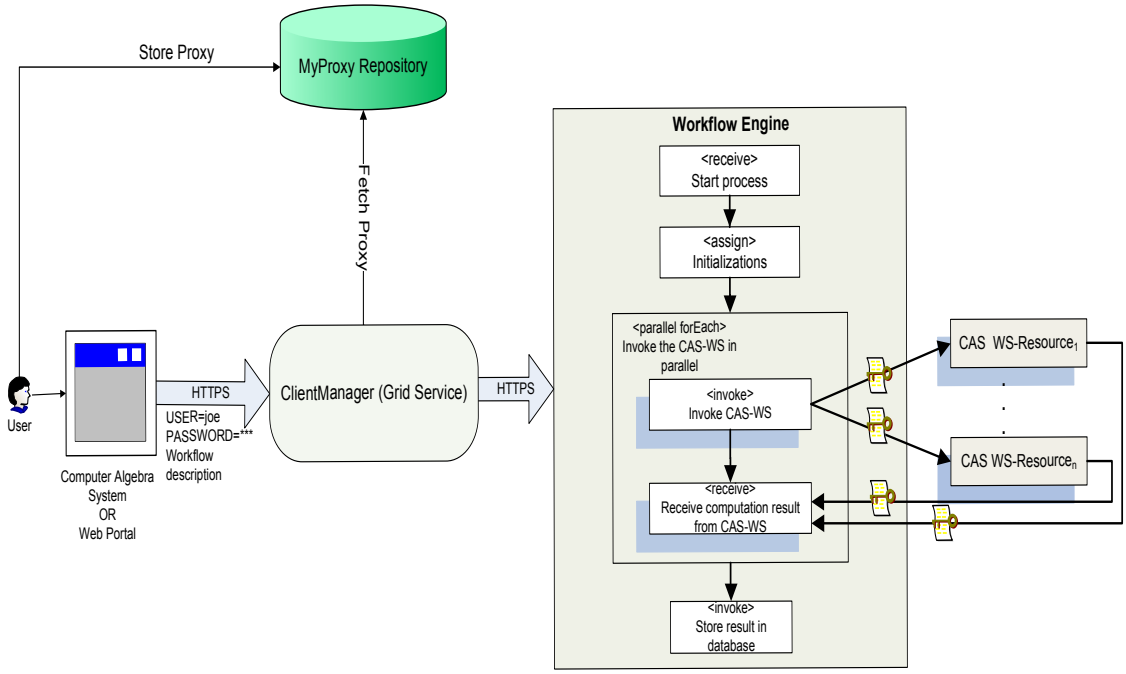


Figure 5.2: Secure Symbolic Components Composition Architecture

a consistent solution. In this respect, it is a matter of user's choice if a certain service is to be trusted or not. Invoking secured Grid services offers a higher level of confidence because the invoker controls the identity of the services being used.

Among its capabilities, CAGS provides support for the service discovery process and access functionality offered by services. Mechanisms to secure Web Services are similar to those for Grid Services as both support security at transport and message level. Web Services implement the security by using HTTP basic authentication and/or HTTPS combined with username and password authentication at the lower level and the WS-Security communication protocol at the upper levels which allows the use of Kerberos tickets and X.509 certificates. For Grid Services, using TLS encryption at transport level and even message level security is a common approach. One important feature that CAGS tool offers support for is security certificates management.

For accessing secured Grid services developed based on Globus middleware, the client component must provide a valid proxy certificate. This requirement is valid for all custom secured services but also by the standard services provided by Globus, such as WS-

GRAM or RFT. The CAGS component is able to manage such certificates and implements required functionality for supplying the certificates to remote services whenever this is required. Part of the steps of the authentication process are not transparent for the user and therefore specific actions must be taken for managing the certificates. An advantage of providing the user specific means to manage certificates is the possibility for the user to switch between different certificates depending on the service the user wants to access. Hiding the management process for the user would be possible with the drawback that the same security certificate would have to be used regardless the remote service accessed.

Secure Symbolic Service Composition

The main responsibility of the AGSSO server is to supervise the execution of symbolic computation workflows submitted by clients. With security features enabled the username and password provided by the user at client side are used by the ClientManager component of AGSSO to retrieve the proxy certificate of the client from the MyProxy manager. The workflow engine uses the user's proxy certificate for communicating with parter CAS Servers. Scheduler components implemented at AGSSO server level have the role to select the most suitable CAS Servers that should be used to execute a certain task. Several criteria are used to determine the CAS Server but for a CAS Server to be selected information about the CAS Server must already be stored in the Main Registry maintained at AGSSO Server level.

The process of populating the Main Registry requires that both the AGSSO Server and the partner CAS Server agree to exchange information about their capabilities. While various information about their state is exchanged in an automatic way, as a first step both AGSSO and CAS Server components must be set to accept this information exchange. At AGSSO level the administrator must register the URLs of partner CAS Servers. Therefore it is the responsibility of the administrator to ensure the CAS Servers accepted as partners are suitable from the security and QoS point of view. Although not

implemented currently, the scheduling algorithms may be instructed to use a separate set of CAS Servers to submit tasks to for certain types of processing, or for requests submitted by particular users. This way, resource partitioning at a fine grained level is possible.

The AGSSO server component relies on ActiveBPEL for the actual execution of workflows. ActiveBPEL engine offers excellent capabilities for orchestrating Web Services but it provides only basic support for security mechanisms through anonymous TSL encryption at transport level. An extension of the ActiveBPEL engine that offers support for accessing GSI secured services was reported in [82]. Since security settings may be applied for every operation of a Grid service interface, the authors proposed to extend BPEL standard language in order to offer support for WS-Security and WS-SecureConversation. These extensions together with some changes of the ActiveBPEL engine enable the engine to apply additional security mechanisms needed to access Grid services implementing security mechanisms supported by GSI.

When a process is invoked, the message is passed to the Globus' security and message handlers added to the the Axis chain of handlers and these handlers automatically encrypt and sign messages. The response from the services is also handled by Axis handler chain which decrypts and checks the received message.

In order to integrate ActiveBPEL with the GSI implemented by Globus Toolkit 4 several configurations must be made to the workflow engine. ActiveBPEL uses Axis and thus one way to enable the ActiveBPEL engine to run a process using the proxy certificate of the user that requested its execution is through Axis handlers. This means that we need to declaratively add some security handlers into ActiveBPEL's message chains. When a part of the process calls an external service, ActiveBPEL plays the role of a Client to that Service. Thus, messages that originate from ActiveBPEL will need all credentials added prior to being sent. In both cases, the correct handlers must be placed in the message chains.

CAS Server Security Features

The CAS Server components allow exposing symbolic capabilities implemented by CASs as Grid Services. The specific operations on the CAS Server's interface represent access gates to CAS provided capabilities that are installed on the CAS Server. Since these services are running in a Globus Toolkit container, all security features are provided by Gobus GSI. Although GSI implements three security mechanisms - GSI Secure Message, GSI Secure Conversation and GSI Transport - we have chosen to use only GSI Transport for initial version and later extend our solution for integrating the other security mechanism. At transport level security in GSI uses public key cryptography and it guarantees privacy, integrity and authentication.

For each new client that requests solving a symbolic computation task, a new WS-Resource is created at the CAS Server level. Therefore security must be enabled at WS-Resource level to guarantee that no other users except the entitled one may access or modify the state of a resource. To configure security at individual resource level, a security descriptor that describes the access policy must be provided when the service is deployed. This descriptor instructs Globus container that only the user who instantiated the resource is authorized to read or to modify its contents.

Because CASs installed on the CAS Server are exposed through the Grid Service interface, a thorough analysis must be done to identify potentially harmful functions implemented at CAS level that should not be available for remote invocations. Authorization policy in effect at the computational node can be enforced by filtering access to CAS's functionality based on the information stored within the Local Registry of the CAS Server. This registry holds information about the CAS systems that are installed on the computational node, about the functions available to be remotely invoked and the users that are entitled to access the functionality.

The decision on who is authorized to access a certain service or computational resource has two important components: functional and legal. At the functional level, a decision

must be taken whether a call must be accepted and executed depending on details such as the effect that the call has on the target system, the level of security needed, the resource utilization, etc. An equally important matter regards the legal right to access a resource. The CAS Server implements functionality that allows administrators to edit information within Local Registry of a specific CAS Server. This functionality is not available through the Grid Service interface but through command line utilities. The access to administrative command line utilities is controlled using proxy certificates for authentication which makes possible implementing a VO wide administration policy.

The informational structure of the local registry system allows to store the following details:

- The name, version, local install path or the CAS
- For every CAS it lists the functions that should be accessible for remote invocations, expected list of arguments, the name of package to be loaded when the functions is called and a short description of the function
- Security details such as the users entitled to access them

The information stored in the Local Registry contains various details that allow the system to restrict user's access to specific functions which they are not entitled to call. Even more, details regarding the machines and software packages installed in the scope of a certain CAS Server may be available for some users and hidden for others. Details regarding which CAS is installed on which physical machine that lies behind the CAS Server's interface are not made public for security reasons. Even more, if particular functions should be accessed only by privileged users, the rest of the users will not even obtain their list during discovery process.

As noted before, the information provided in the Local Registry is synchronized with the information maintained by partner Main Registries implemented by AGSSO servers. At deployment time, when the tasks to be executed are mapped to the actual computational

nodes that are going to perform the computations, the information stored in the Main Registry of the AGSSO server plays an important role. If the AGSSO server determines that a certain CAS Server is able to support a certain functionality it may schedule a task to be solved by the particular CAS Server.

In the case when symbolic Grid services are accessed directly using the CAGS component at client side and not by the AGSSO server, security is also enforced using the proxy certificate of the user. This certificate can be stored locally on the machine of the user, but it could also be obtained using the MyProxy credential repository. After the user is authenticated, the invoked Grid service will check if the user is authorized to call the operation using a special gridmap file.

5.3.3 Conclusions

As a result of our investigation we conclude that security mechanisms provided by standard Grid Services are suitable for establishing a secure infrastructure for symbolic computations. To enforce security in our architecture we rely to a large extend on security mechanisms provided by the Grid middleware and we apply them. Globus GSI provides support for the HTTPS protocol, for the use of Certification Authorities and X509 security certificates. ActiveBPEL execution engine does not provide native support for orchestrating Grid Services but this shortcoming can be overcome by implementing custom extensions.

Additional security is enforced through mechanisms that rely on the specific features provided by the CAS Server and the AGSSO Server components. At both CAS Server level and AGSSO Server level, registries hold relevant information regarding users and resources. Based on this information, system administrators may implement policies that define trusted partner relations, restrict users to access particular features and eliminate security threats.

5.4 Summary

The design and capabilities of the Client Component of our architecture described in this chapter were previously presented in [60, 150]. The security related issues that apply to our overall architecture were previously discussed in [63].

Web and Grid Services represent a convenient solution for exposing computational capabilities to remote clients. The number and the variety of available services has rapidly increased over the last years and represents a viable way of creating complicated applications based on already existing components. In Section 5.1 we describe the design and capabilities of the Client Component. Due to its design the Client Component can be easily integrated within any CAS with a minimum effort. Through its generic subcomponent which is CAS independent, the Client Component allows CASs to access functionality provided by remote Web and Grid Services. The same component provides support for describing and submitting symbolic computation workflows to AGSSO Server components previously described in Chapter 4.

As we show in Subsection 5.1.2 computer algebra specialists do not have to write themselves complicated code that allows them to access remote services. They only have to use the appropriate functions provided by the Client Component to discover remote services, invoke them and retrieve results. Providing this functionality as an external add-on and not directly and fully implemented within the CAS makes the adaptation of the components easier in the case a migration to technologies other than Web and Grid Services.

The CAGS component also implements features that assist the user in the process of describing and managing workflows for symbolic computations (Section 5.1.2). The user can describe such workflows by combining basic workflow patterns and the resulted workflow can be submitted for automatic execution and management by a AGSSO Server. Within this chapter we have provided several examples implemented in GAP that demonstrate the way sequences, parallel executions, conditional patterns and repetitive

patterns may be used. The SWIP package implemented for GAP represents a thin layer required to access the functionality provided by the CAGS component. In a similar way, the same level of support can be provided with minimal effort within other CASs or even other types of environments.

In Section 5.3 we analyse possible security threats that our architecture have to face. For our implementation we rely on security mechanisms provided by Globus GT4 middleware. Among other features Globus GT4 provides mechanisms that allow building safe Grid infrastructures by implementing several security standards. We have analysed potential threats and we have presented a solution for secure access to AGSSO components based on the standard security mechanism that are currently used in Grid environments. The workflow engine that we use as a subcomponent of the AGSSO server is not prepared to access secure Grid Services and small enhancements have to be added to the wokflow engine. At the client side we have developed the required means to allow users to access secured services. Part of these capabilities, the CAGS component is able to manage security certificates that are required for any user that needs to authenticate while accessing secured Grid Services.

Chapter 6

Advanced Management and Fine Tuning

In the previous chapters we have described the main components that represent the foundations of the massively distributed architecture for symbolic computations that we envisage. Based on CAS Servers introduced in Chapter 3 and AGSSO Servers introduced in Chapter 4 complex computational infrastructures may be created. In this chapter we discuss several advanced features related to data management [65] (Section 6.1) and management of workflows [64, 65] (Section 6.2). We also present a discrete event simulation platform that we use for verification and validation testing and as a framework for fine tuning of components, especially components involved in tasks' scheduling [59] (Section 6.3).

6.1 Resolving OpenMath References

This Section describes the general process that must be followed across our architecture for resolving OpenMath references. In Subsection 6.1.1 we describe the general process that we follow for resolving OpenMath references. In Subsection 6.1.2 we present the

different formats accepted to define OpenMath references by our components. In Subsection 6.1.3 we present the process of resolving OpenMath references if encountered in a SCSCP call request while in Subsection 6.1.4 we present structure of the file obtained as a result of the resolution process. A more complex example is given in Subsection 6.1.5 while in Subsection 6.1.6 we explain how the resolution process was integrated with existing data management capabilities provided by Grids.

6.1.1 The Process of Resolving OpenMath References

Management of data in distributed environments requires a careful attention and viable data management policies and constraints have to be enforced. Workflows specific to symbolic computations often require that large sets of data are exchanged between collaborating components. Autonomous components of the system may collaborate by exchanging data which is only possible if compatible data encoding and data management mechanism are used. In particular, the use of OpenMath as data encoding standard requires that the system is able to understand OpenMath references and provide support for resolving such references if they are used for describing OpenMath objects.

The OpenMath standard briefly described in Subsection 2.6.1 provides means to encode mathematical objects in a format that is platform independent and therefore that can be used for communication between mathematical systems even if they do not use the same data encoding model for internal manipulations of mathematical formulae. Representations of mathematical objects can sometimes be lengthy and as a result files holding representations of mathematical objects may be large. Manipulating such large representations requires significant time and computational resources. One possible solution is to eliminate redundant definitions and to split composed objects into multiple files. This can be done by using OpenMath references mechanisms presented in Subsection 2.6.1.

A compound OpenMath object may be defined by referring to objects defined in the same document or even in external documents. For correct manipulation, a CAS parsing

the definition of a compound object must be able in most cases to access all definitions of sub-objects of the compound object. If OpenMath references are encountered while parsing, the CAS has to be able to access referenced definitions from files stored in the local address space of the CAS. If the sub-objects are stored by remote hosts, they have to be made available locally through mechanisms that are out of the scope of the CAS and therefore they have to be provided by third party components, in our case the CAS Server. The CAS Server component provides mechanisms to support the process of resolving OpenMath references as described in the following subsections.

The resolution process is important both at AGSSO server level and CAS Server level where the OpenMath objects are actually manipulated. At AGSSO server level, the selection of a suitable CAS Server to resolve a particular task must take into account as primary criterion the ability of a certain CAS to handle a particular task. A CAS is able to understand a request only if it is able to recognize all OpenMath symbols used to define the object. Therefore, the AGSSO server must know the complete list of OpenMath symbols used to define the task before sending a task to be resolved by a certain CAS Server.

AGSSO server components are not designed to store actual OpenMath object representations. These are stored by various CAS Servers which act as OpenMath objects storage repositories. To support resolution processes that may occur at AGSSO server level or within other CAS Servers, the CAS Server components implement functionality to extract either lists of OpenMath symbols used within a certain object definition or to retrieve the object itself. Given a set of OpenMath references targeting OpenMath objects that are stored as XML documents in the CAS Servers file system, the CAS Server is able to extract:

- The list of OpenMath symbols used in the scope of the target OpenMath objects; this operation is required at AGSSO server level during the process of selecting the suitable CAS Server to handle a task

- The targeted objects stored in a separate file for later retrieval; this process occurs when an OpenMath object is defined based on objects that are stored by other CAS Servers. To parse the object, all required sub-objects must be accessible locally for the CAS to read and interpret them

Whenever a task that is built using OpenMath references must be assigned for execution by a CAS Server several steps must be executed. The initial task description is parsed and all the OpenMath references are identified and extracted. Every OpenMath reference is investigated and references are grouped based on the CAS Server that hosts the referenced objects. For each group of references a call to the corresponding CAS Server is issued for retrieving the list of OpenMath symbols that are used to define referenced objects. If a targeted object contains itself references to other OpenMath objects, the CAS Server hosting the object is responsible for identifying them and it requests further the list of OpenMath Symbols from the respective hosting CAS Servers. This recurrent process generates therefore a chain of calls in which several CAS Servers collaborate for retrieving the list of OpenMath symbols.

The information flow between CAS Servers is acyclic in order to prevent unnecessary data transfer. At a certain step during the execution a chain of CAS Servers is constructed and maintained system wide. Through the messages exchanged by collaborating CAS Servers, each CAS Server is aware of the list of CAS Servers that are already part of the resolve chain and it does not formulate requests to the CAS Servers that are already in the chain. Each CAS Server responds to requests that are formulated by its ancestor in the chain and is able to formulate resolution requests, one at a time, to another CAS Server which is not yet part of the chain. References that are in the scope of ancestor CAS Server are not resolved but sent back as part of the resolve response it formulates. In this way the order is preserved and unnecessary calls that would lead to a cycle are avoided. Any CAS Server receives therefore a set of references that are in its scope and provides as a response a set of symbols that it was able to discover and a set of references that should be handled by CAS Server that have a higher rank in the resolution chain.

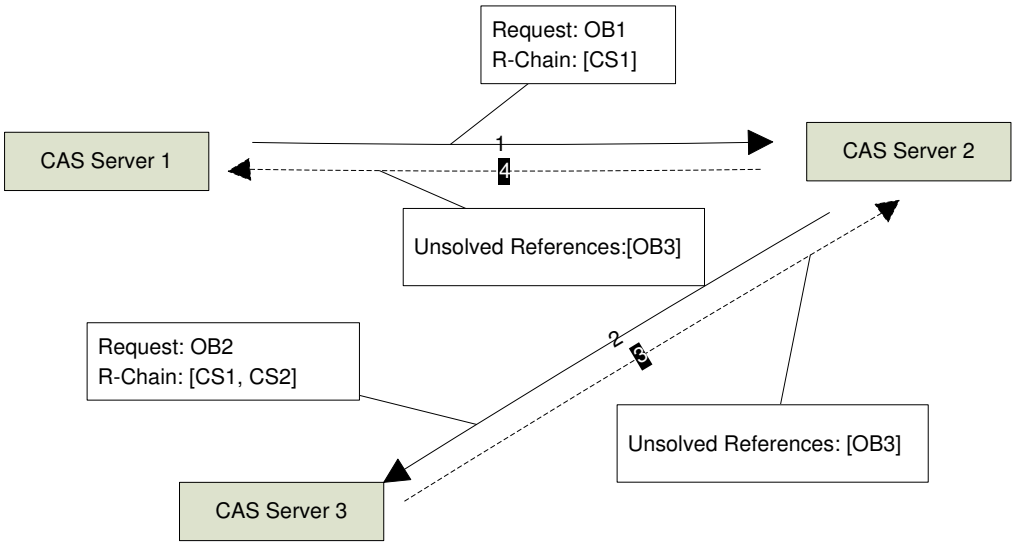


Figure 6.1: Cyclic Data Flow Prevention.

In the Figure. 6.1 we assume that the CAS Server *CS1* discovers while reading a local OpenMath object the OpenMath reference *OB1* referencing a object that is hosted by CAS Server *CS2*. Therefore it sends a request to CAS Server *CS2*. The request contains the reference *OB1* to be resolved and the current resolve chain which contains the CAS Server *CS1*. While reading the object targeted by *OB1*, the CAS Server *CS2* discovers another reference *OB2* to an object hosted by CAS Server *CS3* and it sends a request to *CS3*. The reference to be resolved is *OB2* and the resolve chain that the CAS Server *CS3* receives is [*CS1*, *CS2*]. While reading the object targeted by *OB2*, CAS Server *CS3* discovers that a reference to *OB3* hosted on CAS Server *CS1* has to be resolved. Instead of sending a request to *CS1*, it sends back the result of the resolution operation to *CS@* and a list of references that it can not resolve which contains the reference *OB3*. This reference is not solved by CAS Server *CS2* either, because *CS1* is in the resolution chain for *CS2*. It passes the unresolved reference to *CS1* together with the results obtained so far. This algorithm makes sure that data is not create unnecessary data flow loops and makes possible to identify potentially malformed OpenMath documents since cyclic references are not allowed in OpenMath [184].

Unlike the partial resolution process described before that only requests lists of OpenMath symbols, the resolution process for obtaining the actual OpenMath objects is slightly more complex. The role of this process is to collect referenced OpenMath objects to the machine on which they are going to be parsed by the CAS hosted on the CAS Server. Any task submitted for execution to a CAS Server is described as an initial call that may contain references to objects not hosted by the CAS Server. The first step in the resolution process is to retrieve from the initial task description the list of references. Objects that are local to the CAS Server are extracted from their original files and stored in a single file. References that point to partner CAS Servers are grouped by their hosting CAS Server, and requests are sent to partner servers. A server receiving a node resolve request extracts OpenMath objects that are available locally to a temporary file and a URL representing a download link to the file is sent back to the requesting CAS Server. Similarly with the symbol resolve process, node resolve process constructs resolve chains. The difference is that messages sent back to the requesting CAS Server contain in this case a list of URLs that can be used to retrieve the actual objects as files.

Delaying the actual retrieval of referenced objects to the end of the resolution process improves communication efficiency because the referenced objects do not travel along the resolution chain. They are hosted in temporary files created at hosting CAS Server level and the download URL is only used by the CAS Server that actually needs to collect the OpenMath objects. This CAS Server accesses the remote temporary files and appends the content of those files to the common file where local objects were extracted. At the end of this process, all objects required for execution are therefore located in a single local file from which the executing CAS can retrieve them. Due to additional manipulation of objects and update of references, the resolution process makes sure that the content stored in the resulted file represents a valid OpenMath object in terms of structure and OpenMath references links. The disadvantage of this approach is that it is not able to detect changes that may occur in objects hosted by other CAS Servers while the resolution process is still executed. Once referenced objects are extracted to temporary files at different CAS Server levels they represent stand-alone objects and

they are not kept in synchronization with the originals if the originals are modified after extraction.

There are situations in which the resolution process is not required. This situation may occur if one instance of CAS named *CAS1* requires the support of another CAS installed on a different machine *CAS2* to execute some computation without requiring the actual computed results. In this scenario, *CAS1* sends a request to *CAS2* which results in the creation of object *R*. *CAS1* subsequently requests more computations to be done using *R* as input parameter. In this case it may be that *R* is not needed at *CAS1* site, it may be to large to be transferred from *CAS2* to *CAS1* and even *CAS1* may not able to understand and use *R*. One solution is to use the concept of cookies implemented by SCSCP protocol [95]. The second solution is to use OpenMath references. In both cases as a result of the initial request which creates *R*, *CAS1* receives a SCSCP cookie or an OpenMath reference that identifies the object. In subsequent calls made from *CAS1* to *CAS2* the cookie or reference is provided as part of the request.

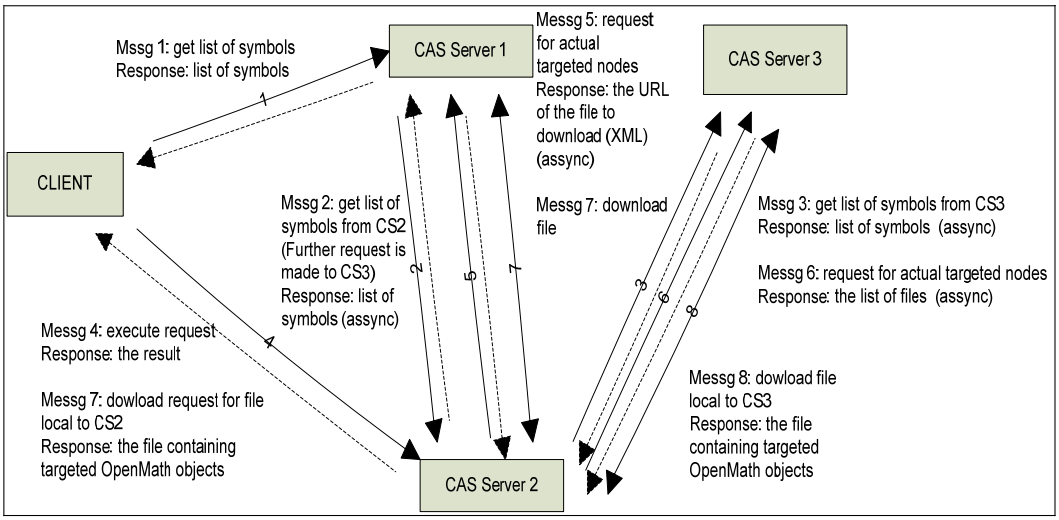


Figure 6.2: Resolving OpenMath References.

To illustrate a slightly more complex resolution process we assume the following scenario depicted in Fig 6.2. The generic client of an AGSSO server submits for execution a workflow composed of multiple tasks. We assume that one of the tasks contains ref-

erences to an OpenMath object hosted by the CAS Server CS1. The object hosted on CS1 also contains the references CS1_OMR1 that targets an OpenMath object hosted in a local document and CS1_OMR2 that targets a OpenMath object hosted by CAS Server CS2. We also assume that the object targeted by CS1_OMR2 contains the reference CS2_OMR1 that targets an object hosted by CAS Server CS3. The symbol resolution chain is thus:

$$CS1 \rightarrow CS2 \rightarrow CS3 \tag{6.1}$$

For the AGSSO server to be able to select a suitable CAS Server to execute the task, it has to determine in the first place the list of OpenMath symbols that it uses. Therefore it formulates a resolve request to CS1. The CS1 formulates further a request to CS2 and suspends its internal resolution process until CS2 contacts back the CS1 server with the response to the request. The response contains the list of symbols discovered by CS2 and any other symbols that were discovered by descendants of CS2 in the resolve chain that are further contacted by CS2, in our case, CS3. At the end of this process, CS1 responds to the resolve request of its client, here the AGSSO server, by sending the list of discovered symbols. Based on the list of symbols that the task contains and the capabilities of the CASs installed on the CAS Servers, the task is assigned to the most suitable CAS Server. For our scenario we assume that the task is sent to CS2.

When the task is received by the CS2, it is parsed and CS2 discovers that the call contains a reference hosted by CS1. It sends a node resolve request to CS1 at which level the object targeted by CS1_OMR1 is copied to a temporary file. The CS1_OMR2 points to a CS2 object so at CS1 level there is nothing to do in this respect. Thus, CS1 responds to CS2 by sending the URL targeting the temporary file and a notice regarding the CS1_OMR2. At CS2 level, the object targeted by CS1_OMR2 is parsed and the system discovers that another object, hosted by CS3 is required. Similar with the previous case, CS2 contacts CS3 and obtains a link to a temporary file. Since there are no other references to be resolved, CS2 contacts CS1 and CS3 and downloads from them the tem-

porary files containing targeted objects. As a result of this resolve process, all objects required for execution are now stored locally to CS2.

6.1.2 OpenMath Reference Formats

The symbol resolution process only tries to discover the list of symbols that are used to describe a certain task. The role of full resolution process is to make sure that all objects required are downloaded and accessible to the CAS that will carry out the execution. A task submitted for execution at CAS Server may contain references to OpenMath objects located in the scope of the CAS Server or located on other CAS Servers. In OpenMath, the general format of a OMR is:

```
<OMR href="URI" >
```

The format mentioned above is flexible enough to accommodate any naming scheme but further restrictions to this format should be imposed to make it effective in the context of distributed processing. For consistency reasons, we have imposed several format restrictions. The accepted formats that the URI can take within OpenMath documents handled by the AGSSO system are the following:

- Absolute URI should be used to designate resources by providing all the information required to locate and retrieve them;
- Relative URIs suitable for identifying resources relative to a certain location previously supplied;
- Local file URIs which are used to fully identify resources that are hosted by the local machine;

The format used for describing absolute URIs is the same with the one used for identifying Web pages. The format of the URI is:

```
http://host:port/path/to/file/filename/#identifier
```

The protocol part of the URI which for our purpose is designates as “http” may be in our case disregarded if better transfer protocols are available. With Grids, other transfer protocols for transferring data such as RFT are used instead of HTTP. The *host* and *port* elements of the URI must identify valid CAS Server listeners that implement the required interface for resolving OpenMath references. A client that needs to access a certain resource will use the host and port information to call the appropriate services. In the context of the server hosting the resource, the path to the actual location of the file containing the targeted resource should be identified by mapping the filepath section of the URI (“path/to/file/filename”). The CAS Server provides an implementation for which all the files accessible through the resolver interface are stored in a common root directory. CAS Server automatically maps the file path to the actual location of the file starting from this root directory. The last part or the URI represents the reference *identifier* used to identify the OpenMath object within the targeted file.

Relative URIs may be used to reference OpenMath objects having as start point the location of the file in which such references reside. Therefore this type of reference can only be used to identify resources that are hosted on the same machine as the document containing them. The general format is :

```
path/to/document/#identifier
```

for which to identify the actual file general rules applicable in Unix and Windows operating systems are used, also valid in the case of relative URLs. If the path to the file is missing and only the “#identifier” part of the URI is present, the current file is assumed.

The support provided by CAS for dealing with OpenMath references is still under development and there is no standard format accepted by all CASs that can be used to identify and extract OpenMath objects based on their URI. GAP is able to retrieve OpenMath objects from locally available XML documents if the format:

```
file:///path/to/file/#identifier
```

is used. Because the format mentioned above lacks information that would allow to identify resources in a distributed environment URIs following the absolute URI format presented above must therefore be transformed before they are sent for evaluation by the CAS.

The resolution process for URIs starting from a request sent to a CAS Server is complex and potentially involves coordination among several partners. Rewriting rules are needed to ensure consistency and correctness of references even after the objects are moved from one hosting CAS Server to another. Replica management may be very difficult if the system has to keep track of various OpenMath objects that are hosted by different CASs. At this stage of development AGSSO does not provide support for replica management as a consequence identical objects that are hosted by two different CASs are considered to be different.

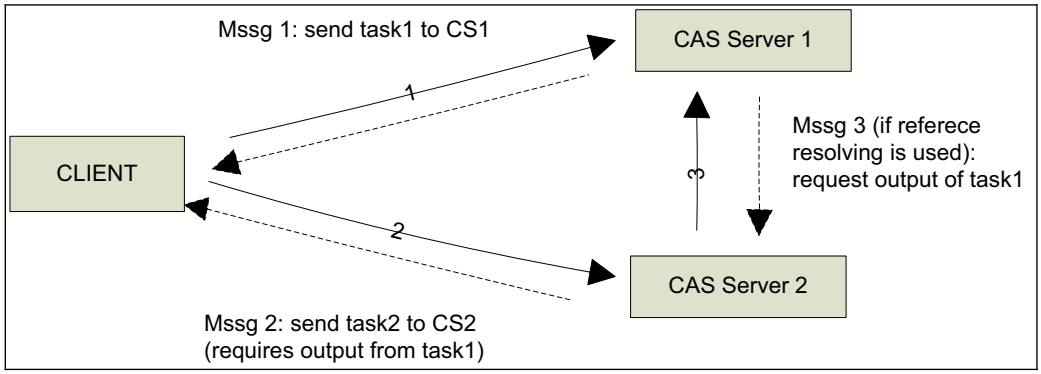


Figure 6.3: Sample Resolver Scenario Architecture.

The use of OpenMath references is not only beneficial in the initial description of tasks. The same mechanisms may be used to minimize network traffic and execution time for sequences of tasks among which data dependencies exist. As a simple scenario we assume that a client of a CAS Server submits for execution a task *task1* to the *CASServer CS1* component and after completion of this task, it sends the *task2* to *CAS Server CS2* which requires the output from *task1*. The most efficient solutions for this situation is to make sure that the response for task1 is not a self contained description of the result, but

an OpenMath object that contains an OpenMath reference to an object hosted by CS1. As input data for *task2* the client does not provide the actual parameters but the absolute reference that are resolved at CS2 level. Before starting the execution at CS2 level, CS2 contacts CS1 and obtains the actual parameters by resolving the OpenMath reference as depicted in Fig. 6.3.

6.1.3 References in the Main SCSCP Call Document

The SCSCP call document describes the call and meta-information related to the call, in concordance with the SCSCP specification. The document has two main sections: the header section where details regarding the call are specified; the body where an OMA OpenMath objects attaches to a symbol identifying the remote operation to be executed, and the objects that represent the parameters of invoked operations.

The CAS Server is able to handle OpenMath references that replace parameters of sub-objects of the parameter objects but references are not accepted for replacing any other section of the call, such as the header of the call. Since the CAS executing a certain computation must be able to identify and retrieve the objects referenced within the call, when sent to be parsed by the CAS this document must contain only absolute URIs. The call document itself is not stored as a file on the machine where the CAS is installed and therefore relative URIs have not associated meaning. Moreover, since the call is received from a remote client, resources can only be correctly identified if they contain complete information regarding the host from which the targeted objects must be retrieved.

During parsing, the original call document submitted by the client is modified and prepared to be submitted to a CAS. All absolute references are resolved and modified to follow the local file URI format described above. All OpenMath objects referenced are retrieved from remote partners and they are stored in a local temporary file from which they may be read by the CAS. During rewriting process all unique identifiers of objects in the OpenMath document are modified in order to avoid naming conflicts. The retrieved objects may in turn contain other references that may point either to the local

CAS Server or to other CAS Servers. These references must also be resolved before the actual execution may start.

References to documents hosted on the local host or even to objects described in the same document are rewritten to relative URIs which point to the same temporary document. Objects that are locally hosted in other files than the temporary one are parsed and inserted in the temporary file. The goal is to minimize the parsing effort that the CAS must do to identify and retrieve objects required for processing. A similar approach is used during the resolve process for references that are resolved on third party CAS Server resolvers.

Absolute references discovered in the resolution process that do not point to local documents must be resolved through requests issued to CAS Servers that host the resources. When the resolver on the execution CAS Server discovers such references during the parsing process, it formulates to the appropriate CAS Server a request that contains: the list of references that the third party must resolve, new identifiers that will replace the identifiers of the targeted objects and a list of absolute root references that the third party resolver must ignore. The identifiers are required because all discovered objects are copied to a single file at the execution CAS Server,and therefore every node must have a unique Id.

6.1.4 The Structure of the Consolidated Resource File

The temporary document containing all objects referenced directly or indirectly by the task call must be a well formed XML document that complies with all OpenMath rules. Because this file contains objects appended during the resolve process, we use a standard OpenMath list specification structure. The basic structure of the file is:

```
<OMOBJ xmlns="http://www.openmath.org/OpenMath"
              version="2.0" cdbase="http://www.openmath.org/cd">
  <OMA>
```

```
    <OMS cd="list1" name="list"/>
    <OMOBJ > ... </OMOBJ >
    <OMOBJ > ... </ OMOBJ >
  </OMA>
</OMOBJ>
```

where the inner <OMOBJ>; ... </OMOBJ> must represent valid OpenMath objects from which the starting and trailing OMOBJ tags are omitted. The resulted document will thus contain a multi-dimensional list for which the depth and dimension depend on the structure of the objects/references that must be resolved.

6.1.5 A More Elaborate Resolution Scenario

We assume that the SCSCP call (incomplete) presented in Listing 6.1 is sent to the CAS Server CS2. At the CS2 level the call is parsed and the "http://cas1.ieat.ro/file1.txt#id1" reference is discovered. This absolute URI must be changed in order to be correctly handled by the executing CAS. Because it is part of the main SCSCP call, the reference is transformed to a local file URI pointing to new local temporary file that will contain all objects obtained through the resolve process. Therefore the reference is modified to "file///local_repository/result_file#newid1". To retrieve the targeted object a call is formulated to CS1 that contains the reference that needs to be resolved and the new ID that must replace the old reference ID.

At CS1 level, the server extracts the object targeted in the reference and stores it in a temporary file with the new XML ID "newid1". All references that are discovered at the CS1 level and start with "http://cas2.ieat.ro/" are skipped from resolving. Now we assume that, in the resolution process at CS1, two references "http://cas2.ieat.ro/file2.txt#id2" and "/file11.txt#id11" are discovered. The targeted objects are copied to the a local temporary file and the two references are changed: "http://cas2.ieat.ro/file2.txt#id1" is changed to "#generatedId1CS1" and "/file2.txt#id2" is changed to "#generatedId2CS1". The ID of the XML targeted nodes are changed, in the temporary file, in our case from

"id11" to "generatedId2CS1". The response of the resolve process at CS1 level informs CS1 that the reference "http://cas2.ieat.ro/file2.txt#id1" was not resolved because it is the scope of CS2 CAS Server. It also specifies the URL of the file hosted at CS1 level which must be downloaded by CS2 : "http://cas1.ieat.ro/tempfile1.txt".

At CS1 level, the message is received and the unresolved reference is resolved by identifying the targeted objects and by copying the objects, with updated identifiers, to the local result files. In a similar way, the rest of references are resolved. When this phase of resolution is completed, as the final step, the file is downloaded and its content is copied to the result file. All its content is copied to the file designated by the reference "file///local_repository/result_file".

```
<OMOBJ xmlns = "http://www.openmath.org/OpenMath" >
  <OMATTR>
    <OMATP>
      <OMS cd="scscp1" name="call_id" />
   <OMSTR>194.102.63.120:26133:6766:dgsyte</OMSTR>
   <OMS cd="scscp1" name="option_return_object" />
   <OMSTR></OMSTR>
    </OMATP>
    <OMA>
      <OMS cd="scscp1" name="procedure_call" />
      <OMA>
        <OMS cd="scscp_transient_1" name="WS_factorial" />
        <OMI>..</OMI>
        <OMR href ="http://cas1.ieat.ro/file1.txt#id1"></OMR>
        </OMA>
     </OMA>
  </OMATTR>
</OMOBJ>
```

Listing 6.1: SCSCP Call with References

Because CASs are not expected to implement data transfer protocols used by various

distributed frameworks, the support for such operations must be provided by external components, in our case the CAS Server component. The interface consists of two operations. One operation allows third party clients to submit object requests while the other one is able to store responses for requests that the current CAS Server previously submitted to other CAS Servers. Through their implemented interface CAS Servers provide support for:

- Handling requests received from external clients for resolving references that are in the scope of the local CAS Server, i.e. hosted by the CAS Server
- Handle response messages for earlier requests that the CAS Server itself has formulated to other CAS Server. This operation is required because the resolve process may take a long time and therefore request/responses are handled using asynchronous messages

6.1.6 Downloading Result Files

During execution all objects referenced by an OpenMath reference and used in the initial call document or in a subsequent referenced object must be available locally. The CAS executing the call should not and is not expected to contact remote machines to transfer the requested objects.

There are often situations when the input data for a task and output data generated by the processing of the task are large. In this situation combining multiple CASs in an execution workflow requires moving this data from/to processing servers that are involved in the computation. A rule of thumb in distributed computing is to minimize the load on the network as much as possible.

The advantages of using OpenMath reference are easily identified. The result of task1 which may not be even needed directly at client side does not have to travel back and forth on the network. Since CS1 and CS2 are server nodes it is highly probable that the

bandwidth capacity of the link between CS1 and CS2 exceeds by far the capacity of the links between the client and CS1 and client and CS2. A collateral advantage of this set-up is that the client requires little computation resources and can be even implemented on hand held devices.

In general CAS lack network communication capabilities and therefore they should not be expected to be able to coordinate transfer of files. CAS Server implements the reference resolver as a separate sub-component of the system and the file transfer is implemented over the functionality provided by Globus RFT (Reliable File Transfer Protocol). Security is also ensured by implemented mechanisms offered by the Globus GSI.

6.1.7 Summary

The resolution process is the process through which references found in an OpenMath document are handled. The components of the system collaborate to provide support for two types of resolution processes:

- Partial resolution is the process to obtain the list of OpenMath symbols used to define a certain OpenMath object. The partial resolve process is required at AGSSSO Server level to allow it to determine CAS Servers that are able to execute a certain task based on the functionality they provide;
- Full resolution is the process to obtain the complete description of an OpenMath object which is defined based on OpenMath references. References used in the definition of an OpenMath object are replaced by the actual referenced objects. Usually the CAS has to have the full definition of the OpenMath objects involved in the execution of a task, including referenced objects.

The AGSSO server and CAS Servers collaborate for resolving references but for retrieval of actual data CAS Servers rely on Grid specific features for data management provided by Globus Toolkit 4.

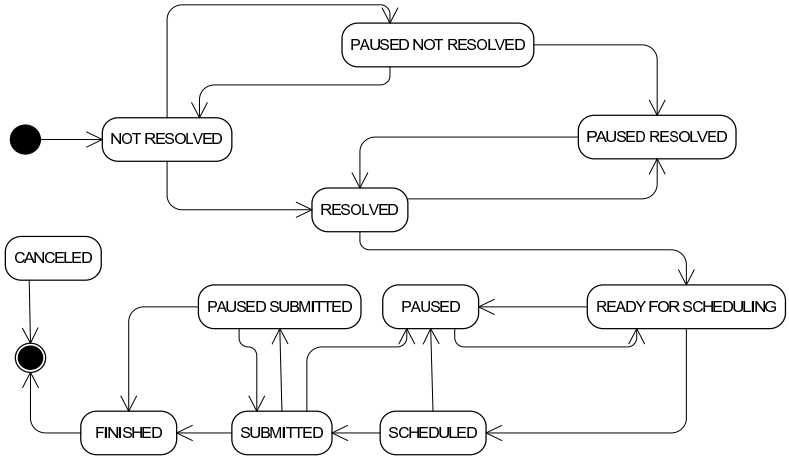


Figure 6.4: Task Life Cycle at Client Manager Level

6.2 Advanced Workflow Management

It is sometimes necessary to intervene in the normal executions of a workflow. Due to various reasons the user may need to cancel or pause the execution and therefore additional management functionalities such as start/stop/resume are required. These actions have a direct impact on the life cycle of the individual tasks from which the workflow is composed of and adds an additional level of complexity for the overall behaviour of the system. In the following subsections we investigate the life cycle of symbolic computations workflows in the way they are handled at the AGSSO Server level and at CAS Server level.

Any workflow submitted by the client to an AGSSO server is managed by Client Manager Component which is responsible for tracking all details regarding computational nodes that are part of the architecture and based on internal rationale to select the most suitable machine to execute a certain task.

The initial status of a task is NOT RESOLVED (Fig. 6.4), since the task described using the OpenMath language may contain unresolved references to external OpenMath objects. Once the resolution process is successfully completed, the status of the task is marked as *RESOLVED* and the task is prepared for further analysis. The task is promoted to the state *READY FOR SCHEDULING* as soon as the task is activated, meaning that all

its predecessors are finished and therefore the tasks can be considered for execution. The system can now run scheduling algorithms to select the most suitable CAS Server to send the task for execution. Once the suitable CAS Server is selected the task is promoted to the state *SCHEDULED*. The following step is to send the task to be resolved to a CAS Server. After the task is successfully submitted for execution, the internal state of the task becomes *SUBMITTED*. Further evolution of the task's state will only occur after the CAS Server submits back the result or an error is returned. Successful completion moves the task into *FINISHED* state while if an error occurs the task is marked as *CANCELED*.

The execution workflow described above is the one that normally occurs if the tasks follows the normal execution path. The user may decide to intervene in the normal execution flow by issuing specific workflow management commands. A 'pause' request may be issued by the client at any stage of the workflow execution which has as effect a change in the current state of task(s) that are in the scope of the request. If a task has the status *NOT RESOLVED* it may be considered that the resolve phase was started but it was not finished yet. Since pausing a task presumably means that the task will be started at a later time it makes sense for the resolve phase to continue. Thus, the task's new status becomes *PAUSED NOT RESOLVED*. If the task is paused while it is in RESOLVED state the new status becomes *PAUSED RESOLVED*. A resume issued for a task in the state *PAUSED NOT RESOLVED* or *PAUSED RESOLVED* will have as effect changing the state of the task to *NOT RESOLVED* or *RESOLVED* respectively.

The parent-child dependency between tasks prevents a task to be scheduled immediately after it is resolved. The task may be *READY FOR SCHEDULING* only when all preceding tasks were already resolved. If a pause is issued while the task is in one of the states *READY FOR SCHEDULING* or *SUBMITTED*, its state may change to *PAUSED*. Any resume operation for a task that is in *PAUSED* state results changing of the task's state to *READY FOR SCHEDULING* that allows the system to run once more the scheduling algorithms to select the CAS Server to which the task should be submitted to.

A submitted task is no longer under complete control of the AGSSO server and any

change in its state has to be coordinated with the CAS Server that the tasks was submitted to. Pausing a tasks in the state *SUBMITTED*, may change the status either to *PAUSED SUBMITTED* or to *PAUSED*. The new state becomes *PAUSED* if the worker was able to pause the task at its level as well, which depends on the status of the task at the CAS Server level and the functionality that the CAS itself provides as described further in this section. If the underlying CAS does not support pausing, the task may continue to run and the state of the task at AGSSO level becomes *PAUSED SUBMITTED*. If the computation ends before the task is resumed the result obtained at CAS Server level is not be sent back to the AGSSO server until the task is not resumed.

One useful behaviour that the system provides support for is discarding the computation of a task and manually assignation of an expected result. The user can reconsider the execution of a certain branch of a workflow or can discard a computation and manually set a result. The later case allows the user to stop a long running task and provide the result of the task without computing it. In such case, by manually assigning a result to a task that is in the state *PAUSED* or *PAUSED SUBMITTED* and resuming the task will have the effect promote the task to the state *FINISH*. No further computation is therefore done for this task and the manually assigned result is used as if it was the result of the task execution. The result that will be used in the rest of the computation is the one provided by the user.

A special action is the cancellation of a task in the workflow. This action may be taken regardless the current state of a task and has a direct effect on all the tasks that depend on the cancelled task. As a result of this action, all descendent tasks, direct or at deeper level, are also cancelled.

At CAS Server level the life cycle of the task is similar to the one at AGSSO server level (Figure 6.5). Once received by the CAS Server the task evolves based on the internal processing and may be influenced from outside by the user's actions. The status of the tasks is promoted from *NOT RESOLVED* to *RESOLVED* if all references were successfully resolved by the system. If the status is *RESOLVED* all OpenMath objects

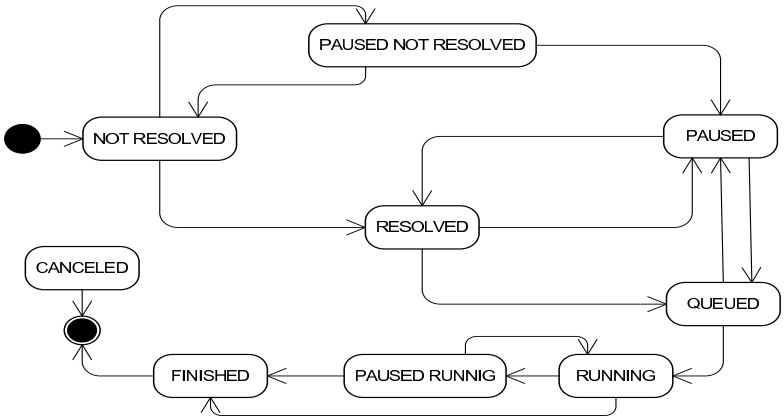


Figure 6.5: Task Life Cycle at Computational Node Level

required for the CAS to be able to execute the task were retrieved from their original hosting CAS Servers. At CAS Server level once the task is *RESOLVED* the internal scheduling algorithm may determine the actual CAS to which the task will be submitted. The CAS may not be able to treat the task immediately and therefore, after scheduling, the task may be put in the state *QUEUED* instead of *RUNNING*. The task is put in the state *FINISHED* when the result was computed by the CAS.

Management actions such as pause/resume/cancel determine corresponding modifications in the state of a task as it happens at the AGSSO server level. A task received by the CAS Server from an AGSSO server is already in the state *SUBMITTED* at AGSSO server level and depending on its current state within CAS Server, its state will change accordingly. A pause requested for a task in *NOT RESOLVED* state will put the task in the state *PAUSED NOT RESOLVED* while if the task is already in *RESOLVED* state it will be put on *PAUSED RESOLVED*. Therefore if the resolve process was already finished it will not be re-executed when the task is resumed. Because it is not possible to freeze the execution of task at CAS level, even if a pause is requested by the user the execution itself continues and from the *RUNNING* state the task is put in the state *PAUSED RUNNING*. The evolution of the task at CAS Server level has also a direct impact on the task's state at AGSSO server level. If the task can be put at CAS Server level in the state *PAUSED* it will also be put in the state *PAUSED* at AGSSO Server level. If the task was already started at CAS Server level and its state is modified to *PAUSED RUNNING*, at

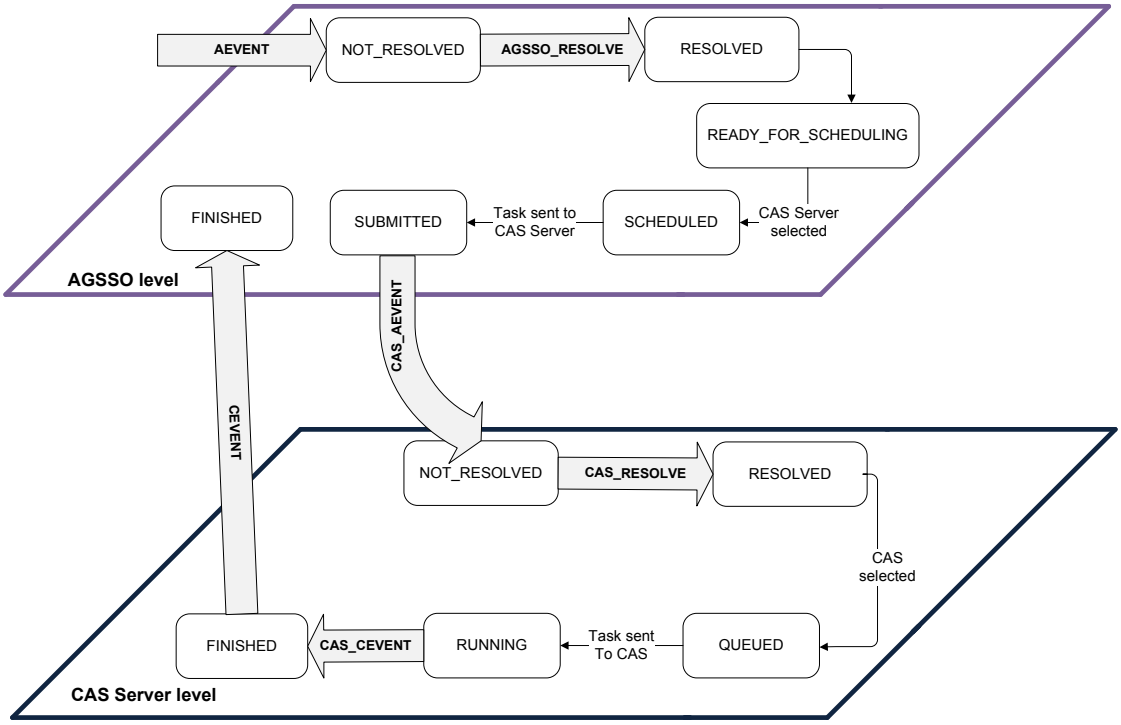


Figure 6.6: The Life Cycle of a Task at AGSSO and CAS Server level.

AGSSO server level the state of the task is set to *PAUSED SUBMITTED*. The correlated relation between the states of the task at system level is depicted in Figure 6.6.

6.2.1 Summary

The life cycle of the task at AGSSO Server level and CAS Server level are similar. The typical life cycle follows the following steps:

1. The task is received by the component, AGSSO Server of CAS Server;
2. OpenMath references are resolved. At CAS Server level full resolve process is applied while at AGSSO Server level only partial resolution is required;
3. The task is sent for execution;
4. The result is obtained.

At user request, the execution of a particular task may be paused or even cancelled, with a direct impact on the state of the other related tasks. The system uses the above mentioned life cycle and information on the current state of the system to determine and control the status of each task. Insight regarding task life cycle is also important for ensuring that the system behaves as expected and as support in development of event-based simulation platforms such as the one presented in Section 6.3.

6.3 Event Based Simulation Framework

In this Section we describe an event-based simulation framework for testing the CAS Server and AGSSO Server components and for providing a test bed environment suitable for optimization of different scheduling algorithms. In Subsection 6.3.1 we present the overall design of the framework. In Subsection 6.3.2 we discuss preliminary results obtained by testing several generic scheduling algorithms.

6.3.1 Simulation Design

Building a distributed architecture is usually not a trivial task as every execution unit in the distributed architecture must act autonomously. The more advanced the implemented functionality is, the less easy is to predict the system's behaviour. In general, testing such architectures in real life environments does not provide a sufficient level of confidence and alternative solutions must be sought. One such alternative is to create a simulated environment. Building a simulated environment for our system offers two important benefits. The immediate one is to validate the implementation by testing not only the separate software components but also their functionality when integrated in the broader SymGrid-Services architecture. Due to the complexity of the system, testing it in the simulated environment helps us to identify problems prior to deploying the system on a real life computational architecture.

A second important benefit of a simulated environment is that it offers valuable information regarding the efficiency of the system. The tasks submitted by a user are analysed by the system and the system tries to find the most suitable computational resources that meet the task's requirements. This is achieved by using scheduling algorithms implemented at two levels of the architecture, as it is described in the following sections. Unfortunately, symbolic computing is atypical with respect to estimating the time required for a task to be completed. Polynomial factorizations offer a relevant example because the time required to factories a polynomial does not vary in a predictable way with respect to the input (e.g. given *P(x)* it is hard to estimate its cost by relying on the cost of *P(x+1)* or *P(x-1)*).

In a real life environment, using prior knowledge regarding the hardware infrastructure on which SymGrid-Services is deployed and regarding the structure of the tasks that are most often resolved by the system, it is possible to do fine tuning of the system in order to achieve greater performance. With such prior knowledge, different scheduling algorithms may be tested for effectiveness and different segregation schemes may be implemented. For efficiency reasons, it is not uncommon in computer farms to apply segregation policies to prevent powerful machines to execute short running tasks or to avoid computational-intensive tasks to be submitted to less powerful computer system.

In the following sub-sections we describe the simulation environment that we have used to test the efficiency of various scheduling algorithms for mathematical problems. The solution uses the discrete event simulation approach tailored to the behaviour and the life cycle of workflows comprising interdependent tasks from the moment they are submitted for execution until their executions ends.

Experiments can be run using various testing and simulation platforms. These include real platforms, simulators and emulators. Real platforms provide better understanding of the system's behaviour in a real time environment but makes testing of different set-ups difficult. Emulators permit flexible testing of existing components by reproducing controlled system calls. If a simulation platform is used, both the components of the

architecture to test and the environment in which it is expected to execute must be created. Simulated platforms represent an usual solution for testing distributed applications under specific circumstances. Examples of such platforms are described in Bricks [173], SimGrid [67], OptorSim [48], GridSim [54], GridNet [120], Wrekavoc [56] etc.

While some of the simulation platforms mentioned above may be used without extensive adaptation in some cases, the particularities of the AGSSO architecture makes reusing already existing simulation platforms a complex task. Therefore we have designed and implemented required components that allow us to simulate the behaviour of AGSSO using the discrete-event simulation mode. Ore goal is to reuse as much as possible of the actual components of the AGGSO architecture. Workflows that would be specified at client side are replaced by generated ones using custom implemented random generators. The structure of generated workflows, the structure of the tasks and the resources they require to be resolved are influenced by configuration parameters of the generators. Therefore a wide range of possible scenarios can be tested without actually executing tasks.

Our simulator aims at mimicking the flow of events that occur inside SymGrid-Services from the moment a workflow is submitted and until its completion. To fulfil this goal, the tasks themselves do not need to be executed by a CAS since from the simulation point of view only details such as time needed for the task to be sent to the CAS Server, time needed for potential OpenMath references to be resolved time needed for the task to be completed and time required to transfer the result are relevant. All these values are actually generated by the simulation platform.

Based on the two-level architecture of the composition framework we have identified several events specific to the simulation environment which have to be processed:

- **AEVENT** - signals the arrival of a new submission in the system. Typically this is represented by a new workflow. The submitting client needs to be registered in the system's database.

- **AGSSO_RESOLVE** - marks the end of the OpenMath resolution process of one or more workflow tasks. This means that the system has identified the CAS servers able to execute the tasks. Once this step is accomplished the scheduler component at the AGSSO level finds the best server for executing the tasks. No rescheduling is possible at this level and thus selecting the optimal server plays a vital role for the global computation costs.

- **CAS_AEVENT** - represents the moment when a task is received by the CAS Server and placed inside its waiting queue.

- **CAS_RESOLVE** - is similar with the corresponding *AGSSO_RESOLVE* event. The only difference is that at this stage the task symbol or method reference links are replaced with the actual content that is transported from the remote hosts to the CAS Server. Once this step is completed the scheduler at server level is started in order to load balance the usage of available CASs. Additionally, a number of tasks can be started on each CAS depending on the maximum number of instances each CAS can handle and on the number of already running ones. Rescheduling is achieved each time this event is triggered. The reason is because each new resolved task needs to be scheduled on the least loaded CAS in order to start executing it as soon as possible.

- **CAS_CEVENT** - is triggered each time a task has completed its execution. The result is stored in a database and the response is sent back to AGSSO;

- **CEVENT** - is triggered when the result from the CAS is received by AGSSO. All tasks depending on the current one are inspected and possibly activated for scheduling.

The events that occur during simulation are closely linked with the actual states of tasks handled by the system. For instance a CAS_EVENT that the simulation platform must handle means that a particular task was finished and the status of the task has changed to FINISHED. While in a real life environment events occur due to normal evolution of

the system's state, i.e. a CAS_EVENT occurs because a CAS finished resolving a task, in the simulated environment the next event is calculated by selecting the event having the minimum due time. The algorithm presented in Listing 6.2 describes the way the simulation platform executes the simulation process.

Simulation starts from the reference time "0" and the current time is increased with each new event that is handled. Initially, because no tasks were previously generated, the system will generate a workflow and the new reference time of the simulation platform becomes the time of the AEVENT. Each workflow and within the workflow each task is defined by the time needed for it to change its state from the initial state to the final state when the task is resolved. If the task's execution started at moment T and it requires N time units for it to be completed, the simulation platform will change its state to FINISHED when reference time becomes T+N. Next event that occurs is always determined as the minimum due time of any event active within the simulation platform.

```
1.     INPUT:  totalNumberOfClients
2.     arriveEcart := 0
3.     WHILE (nrOfRequests < totalNumberOfClients
4.                          OR NOT(finished workflows)) BEGIN
5.         nextEvent  := getNextEventType()
6.         IF (nextEvent = AEVENT) BEGIN
7.             handleAEVENT();
8.             eventTime := currentTime
9.             arriveEcart := generateNextWorkflowArivalTime()
10.            nrOfRequests := nrOfRequests + 1
11.        ELSIF (nextEvent = AGSSO_RESOLVE)
12.           handleAGSSO_RESOLVEEvent();
13.        ELSIF{nextEvent = CAS_AEVENT}
14.            handleCAS_AEVENTEvent();
15.        ELSIF(nextEvent = CAS_RESOLVE)
16.            handleCAS_RESOLVEEvent();
17.        ELSIF(nextEvent = CAS_CEVENT)
18.            handleCAS_CEVENTEvent();
19.        ELSIF(nextEvent = CEVENT)
20.            handleCEVENTEvent();
21.        ENDIF
22.    END WHILE
```

Listing 6.2: Event Based Simulation Algorithm

As described by the Listing 6.3 for the purpose of simulation each task is defined by several characteristics: the amount of memory required for it to be executed; the size of data that must be transferred to the execution host; the list of OpenMath symbols and methods it contains; the time required to resolve OpenMath references it contains; and its relation with other tasks of the workflow. All these details are generated automatically to match specific statical distribution models. Their values may be expressed directly as units of time needed for a certain processing to be completed or as orders of magnitude that influence the time required. Some details such as estimated execution and completion times on a certain resource and transfer costs to and from the server can

only be determined when the *AGSSO_RESOLVE* event occurs. These details cannot be known prior to the selection of the server, a procedure which takes place after the task is resolved at AGSSO level.

```
BEGIN handleAEVENT()
    $n := generate_number_of_tasks();
    FOR (i = 1..n} BEGIN
        nSymb := generate_no_symbols();
        nMet := generate_no_methods();
        mem := generate_memory_req();
        parents := generate_parent_tasks();
        size := generate_size();
        res := generate_resolve_time();
        generate_task(nSymb,nMet,mem,size,parents,res);
    ENDFOR
    FOR (generated_tasks()) BEGIN
        taskState := NOT_RESOLVED;
        insert_task_in_database()
    ENDFOR
   generate_next_workflow_arrival_time();
END
```

Listing 6.3: Generation of New Tasks

The algorithm described in Listing 6.4 shows the actions that have to be taken by the simulation platforms when a task changes its state to RESOLVED at AGSSO server level.

```
BEGIN handleAGSSO_RESOLVEEvent()

    FOR ( ready_to_be_resolved_tasks ) BEGIN
        resolve_task()
        taskState := RESOLVED;
        eet := generate_execution_time();
        tc := generate_transfer_costs();
        update_task();
        taskState := READY_FOR_SCHEDULING;
        schedule_tasks();
        taskState := SCHEDULED;
    ENDFOR
END
```

Listing 6.4: Handling Resolved Tasks

Tasks that are ready to be executed are analysed by scheduling algorithms and the most suitable CAS Server is elected to execute the task based on its requirements and on resources that a certain CAS Server provides. At this level various scheduling algorithms may be tested and their efficiency for a certain configuration may be determined. Once the CAS Server is elected, the task is sent to the CAS Server and the resolve process at CAS Server level is started, following the procedure presented in Listing 6.5.

```
BEGIN handleCAS_AEVENTEvent()

    FOR ( ready_to_arrive_tasks ) BEGIN
        taskState := NOT_RESOLVED;
        res := generate_resolve_time()
        insert_task_in_database()
    ENDFOR
END
```

Listing 6.5: Handling Tasks at CAS Server Level

Depending on the loading of the CAS Server a certain task may be put in a waiting queue

or the task is immediately submitted to the CAS for execution as described in Listing 6.6.

```
BEGIN handleCAS_RESOLVEEvent()

FOR ( ready_to_be_resolved_tasks ) BEGIN
    resolve_task()
    taskState := RESOLVED
    schedule_task()
    $taskState := QUEUED
ENDFOR
FOR (<can start> tasks) BEGIN
    taskState := RUNNING
ENDFOR
```

Listing 6.6: Task Resolution at CAS Server Level

When the tasks is completed at CAS Server level its state is changed to FINISHED (Listing 6.7). The result is sent back to the AGSSO server and depending on its size, its transfer requires a certain amount of time units.

```
BEGIN handleCAS_CEVENTEvent()
    FOR(ready_to_complete_CAS_tasks) BEGIN
        taskState := FINISHED;
    ENDFOR
END
```

Listing 6.7: End of Task Execution at CAS Server Level

Once a task has finished and the result is received by AGSSO server task's state is changed accordingly (Listing 6.8). As a result, other tasks that depend on the finished task completion may further be marked as ready to be analyzed and started.

```
BEGIN handleCEVENTEvent()
    FOR ( ready_to_complete_AGSSO_tasks) BEGIN
        taskState := FINISHED
    ENDFOR
END
```

Listing 6.8: End of Task Execution at AGSSO Level

SymGrid-Services relies on statuses to handle tasks during the entire period from workflow submission to completion. Two sets of statuses are used: one for the AGSSO level (see upper part of Figure 6.6) and one for the CAS server level (see lower part of Figure 6.6). In order to address this matter the simulator's events directly handle the status management as shown by the algorithms described above. Each of these algorithms is called when the corresponding event occurs during the simulated execution.

6.3.2 Simulation Results

One of the main purposes of the simulation platform is to better understand the impact of a real life architecture and the use of different scheduling algorithms used at *AGSSO* and *CAS Server* levels. The different structures of the workflows and the size of the tasks comprising the workflows are two of the important details that should be considered when various scheduling algorithms are used. The result presented here were part of a study previously published in [59].

The first configuration used for testing is the same as the real life hardware architecture that brings together the computational clusters used currently by the SCIEnce project [19]. The two clusters, the one in Timisoara, Romania and the other in St. Andrews, United Kingdom are homogeneous with regard to the hardware profiles and in the software capabilities installed on the machines. In this first set-up, the AGSSO component is installed on the SCIEnce cluster based in Timisoara. The two clusters also act as

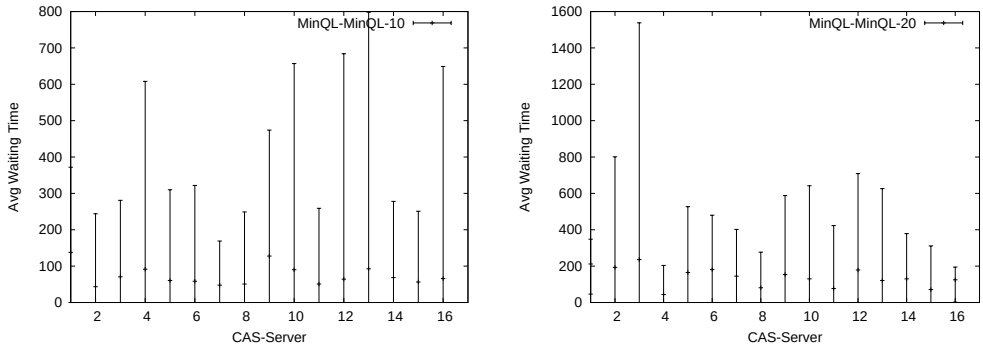
CAS Server components with CASs installed on the nodes of the cluesters. Therefore, we have two CAS Servers, which in turn have eight computational nodes each. Each node of the cluster hosts one instance of GAP respectively.

For testing purposes we have selected three scheduling algorithms: MinQL [98], MaxMin and MinMin [164]. MinQL algorithm ensures that tasks starvation does not occur since the age of the task is considered as a selections criterion. Tests run with this algorithm also use as selection criteria the CPU speed of machines. MinMin first computes the fastest estimated completion time for each task on every resource and it assigns the task to the resource where it would be computed in the shortest time. The MaxMin algorithm is similar to MinMin except for the fact that it assigns the longest estimated running task to the resource for which the value was obtained. The aim of MaxMin is to balance execution of task requiring a long time to complete with tasks having shorter ones.

At AGSSO level only MinQL is used while at CAS server level any of the listed algorithms can be chosen. The reason for this approach is that at AGSSO level there is no rescheduling the target being to balance the number of tasks on the existing CAS servers. At CAS server level we require periodical rescheduling as some CAS could execute tasks faster then others. In our tests we have considered several parameters that would help us to draw conclusions related to the computed results. The makespan represents the total time of execution, from the first moment when a task arrives in our system to the moment when all workflows expected to be simulated by the system are marked as successfully completed.

During the execution, the scheduling algorithms may find all computing resources busy so the task is put into a waiting queue. An important indicator in this respect is the average waiting time related to the executed tasks. We must note here that, due to internal considerations of scheduling algorithms, it is possible that the task is stored in a waiting queue even if a free server is available. This should not though occur often and waiting time should be in this case small in comparison with the average execution time.

Usually, scheduling algorithms try to assign tasks to computing severs so the average



(a) 10 workflows (b) 20 workflows

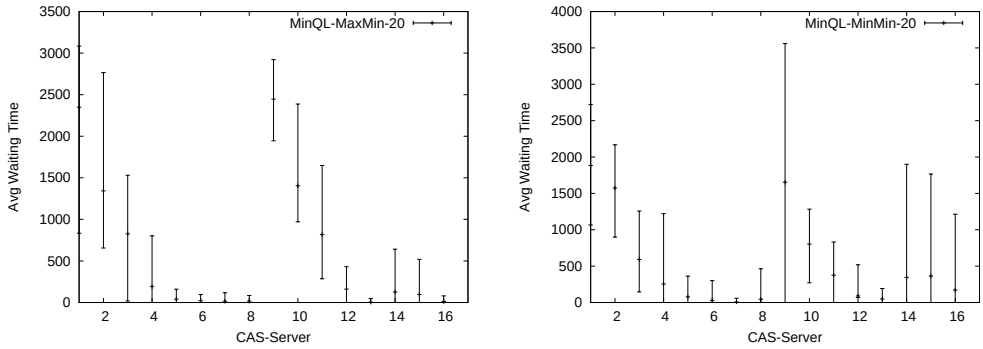
Figure 6.7: Average waiting time for each CAS when the MinQL scheduling algorithm is used at both levels.

load for every machine is balanced. As the following diagrams will show, it is possible that some servers have a greater load due to the structure of the generated workflows. For our purpose we use workflows that combine several execution patterns which may directly affect the load profiles. When dealing with workflows containing sequences, it may be possible that the same machine executes all the tasks of the sequence.

Since the simulation is based on the next-event model, the units of time used in the figures are abstract. For the load diagrams the values used on the Oy axis represent sub unitary values obtained by dividing the execution time of a given server to the total running time. If we consider that tasks at both AGSSO and CAS Server level are scheduled with MinQL algorithm and we submit ten respectively twenty execution workflows the average waiting time in our simulation is relatively small for all servers as can be seen in Figure 6.7. This demonstrates that the scheduling algorithms behave as expected and that the values are similar for the two cases.

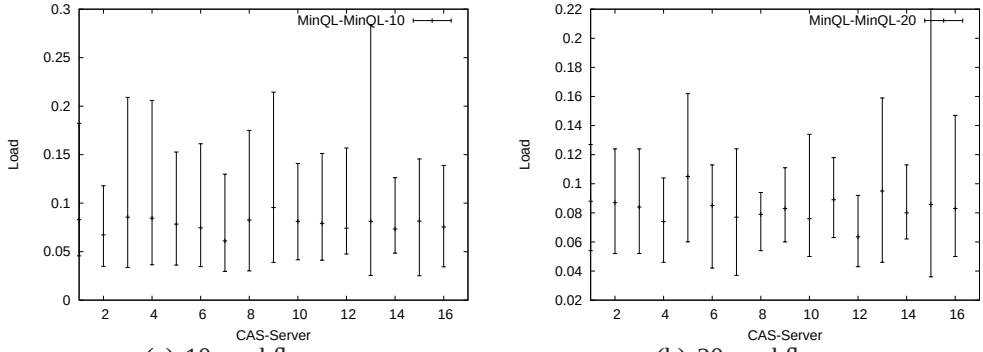
When using different scheduling algorithms at CAS Server level we notice a slight modification in the average waiting time profile (see Figures 6.8 and 6.7(b)). This is due to the fact that MinMin and MaxMin are not load balancing algorithms. This results in higher average waiting time for certain servers.

As we can observe from Figure 6.9 the load when MinQL algorithm is not affected by the number of workflows executed and load between executions CAS is balanced. Not



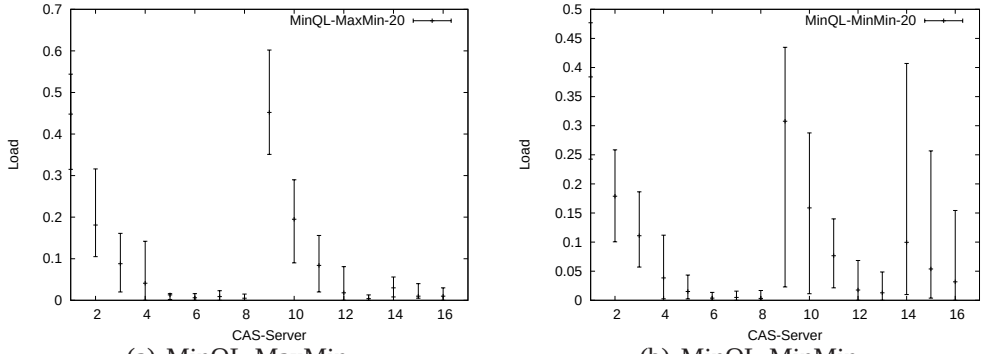
(a) MinQL-MaxMin (b) MinQL-MinMin

Figure 6.8: Average waiting time per CAS for 20 workflows when different scheduling algorithms are used at the two levels.



(a) 10 workflows (b) 20 workflows

Figure 6.9: Average load for each CAS when the MinQL scheduling algorithm is used at both levels.



(a) MinQL-MaxMin (b) MinQL-MinMin

Figure 6.10: Average load per CAS for 20 workflows when different scheduling algorithms are used at the two levels.

the same conclusion can be drawn from the situation when we use MinMin or MaxMin algorithms. These two algorithms led unbalanced loads of the CASs (see Figure 6.10).

Workflow no.	MinQL-MinQL	MinQL-MaxMin	MinQL-MinMin
10	22758 ± 6026	$\mathbf{21230 \pm 5425}$	24397 ± 5092
20	42247 ± 5398	37935 ± 8327	$\mathbf{37450 \pm 6261}$

Table 6.1: Makespan comparison.

Table 6.1 depicts the average schedule makespan (including the standard deviation obtained from the tests). It can be noticed that when MaxMin and MinMin are used at the CAS Server level the obtained makespan is better than the case when MinQL is used at both levels. MinMin and MaxMin algorithms use task estimates when taking scheduling decisions while MinQL focuses on load balancing.

6.3.3 Conclusions

We use the event based simulation platform presented in this section to evaluate the correctness in execution of the two most important components of our architecture, the CAS Server and the AGSSO Server components. We also demonstrate that it is possible to use various scheduling algorithms at both CAS Server and AGSSO Server level. Although the scheduling algorithm used in this section are not specifically tailored for handling symbolic tasks, more efficient algorithms may be developed and tested using the event based simulation platform.

6.4 Summary

This chapter presents several novel features of our architecture, and these have been published as follows. The basic concepts for data management based on OpenMath references and the design and functionality for workflow management are reported in [65]. The design of, and results from, the distributed CAS simulation platform presented in this chapter are reported in [59].

In Section 6.1 we describe how collaborating CAS Servers can resolve OpenMath references encountered while parsing an OpenMath document. We have defined a pattern for expressing references based on the OpenMath standard and we have defined a set of algorithms that minimise the computational resources required for resolving OpenMath references. We have also described a set of components that rely on Grid Services for transferring data between computational nodes.

Section 6.2 describes the process of managing the execution of workflows. In the context of our system we have abstracted the execution process and we have identified the generic states that a symbolic computation task may attain. The identified life cycle takes into consideration the use of OpenMath references for data management and captures the steps required to retrieve the definition of a compound OpenMath object even if its definition is dispersed over multiple hosting nodes. The life cycle also captures the behaviour of the system and steps that need to be executed at various levels of the architecture if execution management capabilities are used, such as pausing, resuming or cancelling a task or even an entire workflow.

Section 6.3 describes a discrete event simulation platform designed to verify and validate the system. In a real environment actions executed by the AGSSO Server and CAS Servers are triggered by specific events that occur, e.g. receipt of a new workflow to execute; tasks resolution completes; tasks are submitted for execution to CAS Servers; tasks execution finishes. The platform receives as input workflows composed of tasks and executes the steps required for execution of the workflows except the actual execution of tasks. Event based simulation platform is appropriate for testing and fine tuning of scheduling algorithms. To demonstrate this functionality we have run the simulation platform with different scheduling algorithms installed at AGSSO and CAS Server level, namely the MinQL, MaxMin and MinMin algorithms. We find that, because MinMin and MaxMin are not load balancing algorithms they induce unbalanced loads, and hence the average waiting time is higher with these algorithms.

Chapter 7

Conclusions and Future Work

This chapter summarizes the main achievements of the thesis (Section 7.1), discusses the limitations of the work (Section 7.2), and outlines some potential solutions that could be provided by further research (Section 7.3).

7.1 Summary

Algorithms for symbolic computations are often complex and they may required a long time to complete. The amount of data they process or generate may also be considerable. Latest advances in distributed computing may provide the required computational resources to support the requirements that symbolic computations raise. Computational Grids represent one of the possible technologies that may be considered for building a computational infrastructure due to several immediate advantages: it provides standard support for data management; it provides standard mechanisms for aggregating and managing resources; it ensures security of the shared resources by implementing well established security policies and mechanisms.

Requirements for an infrastructure that would provide the resources to support symbolic computations field were first investigated more than two decades ago (Section 2.1). In

order to develop a platform for symbolic computations, we have analysed the capabilities and constraints of various architectural styles and distributed computation technologies and based on our analysis we have come to the conclusion that Web Services and Grid Services are the most suitable current technologies for building a distributed computational infrastructure for symbolic computation (Section 2.3). Among the most important problems that have to be addressed is the lack of interoperability between various systems for symbolic computation. Using a common encoding data model for exchanging data between various systems, such as OpenMath, represents an important step ahead towards interoperable systems.

The CASs represent the main tools for symbolic computations and the level of expertise and complexity of these systems makes them valuable and impossible to be replaced or reengineered. To enable these systems to be used as part of massively distributed execution environments, these systems have to be enhanced and additional support components have to be implemented. In Section 3.2 we analyse the most important requirements that should drive a generic interface to expose CASs functionality to be available for remote invocations. Several architectural styles are considered in Figure 3.1 and based on our analysis the server centric architectural style is the most suited to be used as a model for developing an infrastructure for symbolic computations. In Section 3.3 we describe our solution for exposing multiple CASs through a unitary interface exposed using Grid/Web Services. Its architecture is depicted in Figure 3.2.

The role of the CAS Server component is to provide a consistent interface through which functionality of existing CASs can be exposed as Grid/Web Services. Using CAS Servers as foundations, the AGSSO servers provide capabilities to orchestrate multiple CASs for solving compound symbolic computation problems. As depicted in Figure 4.4, AGSSO has the role to manage the execution steps of symbolic workflows and to discover the most appropriate resources for solving a particular problem. To achieve this goal, the AGSSO server combines state of the art capabilities provided by workflow management engines with specially designed components that offer support for scheduling, data management and discovery of resources.

The design of the AGSSO component considers the differences between business workflows and the workflows for scientific computations described in Subsection 4.1.1 and provides a set of capabilities to address these differences. While business workflows are usually composed of a small number of short running tasks, scientific workflows are different. Their tasks are long running and the number of cycles that have to be executed is usually high. Therefore, efficient management of such workflows cannot be achieved without capabilities to control and steer their execution. Features that enable the user to pause/resume/cancel or to alter values of computations while the workflow runs are of paramount importance. The general lifecycle of a workflow and the impact of workflow management actions on an executing workflow are discussed in Section 6.2.

The role of the Client Components of our architecture is to assist the user with description of symbolic computation workflows or while accessing functionality of remote Web/Grid services. The Client Component depicted in Figure 5.1 does not require that fundamental changes are made within existing CASs. Its role is to provide a versatile solution for accessing remote Web and Grid Services from within CASs native environment. It also provides support for describing workflows for symbolic computations as compositions of basic workflow patterns described in Section 4.2. The process of describing workflows is simple and intuitive on one hand, and powerful enough to cover most of the expected computational scenarios on the other hand.

The main data encoding model that our components use to exchange data is OpenMath. Its capabilities to encode semantic rich mathematical content and the associated XML format makes OpenMath the most suitable choice for encoding mathematical content. One of the features that OpenMath provides is the mechanism of OpenMath references. Currently there is little support provided for managing OpenMath references. In Section 6.1 we describe a set of software components that support solving OpenMath references in the context of distributed environments.

Finally, to assist the process of fine tuning for various components of the system we have developed a simulation platform. Using the simulation platform we have made

initial investigations towards refining scheduling algorithms used at AGSSO server level and CAS Server level. In Section 6.3 we have presented the basic algorithms used to simulate the execution of our framework. For testing purposes we have considered so far the implementations of the MinQL [98], the MaxMin and MinMin [164] scheduling algorithms. The results were presented in Figure 6.7, Figure 6.8, Figure 6.9 and Figure 6.10. It can be noticed that when MaxMin and MinMin are used at the CAS Server level the obtained makespan is better than in the case when MinQL is used at both levels. MinMin and MaxMin algorithms use task estimates when taking scheduling decisions while MinQL focuses on load balancing.

As a result of our research we conclude the following:

1. An infrastructure for symbolic computations has to rely on CAS provided capabilities because CASs are the most advanced software tools for solving symbolic computation problems. Functions implemented by CASs have to be made available for remote clients;

2. The most appropriate technologies to use for exposing CASs' functions for remote invocations are Grid and Web Services. These technologies are suitable because they provide standard mechanisms for advertising services which facilitates the discovery of new services, they have a standard data encoding model which relies on XML, and they are platform independent. Web and Grid services are accessed using HTTP/HTTPS protocol which raises less security concerns and is usually allowed by standard security policies. Additionally, Grid Services provide standardized security capabilities which raises the overall security of the computational system that uses them;

3. The structure of the interface exposing CASs must be consistent over time and must provide at least the following mandatory capabilities: a single operation through which remote clients may submit symbolic computation requests irrespective the functions and CASs that executes the requests; a set of operations that

support the discovery process; non blocking mechanisms to retrieve computed results. More advanced functionality that allows task level management for pausing, resuming or cancelling tasks may also be beneficial;

4. One of the most versatile languages orchestrating Web Services is BPEL. Several execution engines that use BPEL exist. ActiveBPEL is one of the most popular open source execution engines and it can also be extended to orchestrate Grid Services. Other existing software tools for describing and execution of Grid Services exist but they can not be easily integrated with existing CASs. Our AGSSO Server relies on ActiveBPEL for managing workflows but it provides additional features such as automatic workflow generation, task management and support for provenance and reproducibility;

5. Provenance and reproducibility of scientific results are of paramount importance for validating research. Data captured by executing CAS Servers and AGSSO Servers allow us to construct a detailed picture of the steps executed as part of a workflow. Therefore we can document any workflow execution and based on gathered information the workflow can be rerun.

6. Data management should rely on existing data exchange protocols and technologies. We have developed algorithms and software components to assist in the process of resolution of OpenMath references which makes data management easier and more efficient.

7.2 Limitations

The architecture that we propose within this thesis has several limitations that are partially related to specific implementations of the CAS Server, AGSSO Server and Client Component and limitations that are related to the functionality that current CASs provide. Within this section we address these types of limitations.

The Client Component currently supports interactions with any type of Web Services but it only supports Grid Services implemented using the Globus 4 framework, which are WSRF compliant Grid Services. Previous Globus Grid Services such as the ones that are implemented using Globus 2.4 or Globus 3 are not supported. Therefore, the client component can not access Grid Services for symbolic computations that are not implemented as Web Services or as Globus 4 Grid Services. Similarly, Grid Services that are implemented using gLite middleware cannot be used as part of the current architecture.

At client side the CAS specialist may describe workflows for symbolic computation based on services that are exposed by CAS Servers. Even if Web or Grid Services implemented by third party providers may be invoked using the Client Component, these cannot be used as part of the automatic workflow execution. As described in Section 4.4 the AGSSO Server can only compose services exposed by CAS Server components. Additionally the implementation to support the differed choice execution pattern is not fully tested and due to lack of reliability it was not included in the set of features implemented at client side.

Although the support for OpenMath is increasing, OpenMath is not yet fully supported for encoding mathematical content within all CASs. Although we provide a workaround for building symbolic computation infrastructures using CASs that do not support OpenMath, the full range of functionality is only available if the mathematical content is encoded using OpenMath. At client side as well as at CAS Server level our architecture relies on the support that CASs provide in this respect.

Scheduling and discovery play an important role in the correctness and overall efficiency of our architecture. At AGSSO Server level and CAS Server level scheduling algorithms are used to select the most suitable resources to be used for running tasks. So far we have investigated the behaviour of the system based on several algorithms but these algorithms are not specifically tailored for symbolic computations. Therefore the scheduling strategies should be improved to match the specific profiles and requirements of symbolic computation tasks.

Currently the system does not fully provide support for pausing and resuming the execution of tasks that fully preserves already finished computation steps. To provide this functionality the CAS engine executing the task should be able to save the current state of execution and to resume it later. Depending on the state of the task that needs to be paused the status of the task is changed by the system and some of the intermediary results are discarded.

7.3 Future Work

The experience of the last decades shows that technology and best practices in the distributed computations world evolve at an accelerated pace. It is therefore important to determine a set of requirements, constraints and models for symbolic computations that are as much as possible independent of the underlying technology. Within this thesis we have analysed the most important requirements that symbolic computations raise and we have designed a set of components and related algorithms that are to a great extent independent of the actual distributed technologies used for interconnecting the implemented components. The architecture we have designed was implemented by relying on existing best practices in Grids. Further research to evaluate our architecture in the context of other technologies as the ones used in cloud computing may provide additional insights.

Establishing repositories of precomputed results may have a significant impact on the time required to conduct research experiments in certain areas of symbolic computation. Currently our implementation does not support querying of such repositories which by itself does not require fundamental changes in our implementation. One simple approach for establishing computed result repositories is to compute every such required result and stored it so it can be later reused. A more complex and also more valuable solution is to establish mathematical equivalence between problems on one hand and obtained result on the other hand. Equivalence of mathematical objects may be sometimes difficult to determine and further research is required both in the area of mathematical equivalence

and with regard to equivalence of algorithms. For instance the system should be able to automatically detect if two workflows are equivalent even if they are expressed in a different form.

The irregular nature of symbolic computations algorithms makes prediction of required resources and completion time difficult. Therefore, more suitable scheduling algorithms that are able to consider characteristics specific to symbolic computations have to be developed. This goal can only be attained as a result of a long term and careful monitoring of the execution patterns, types of task of symbolic nature and underlying CASs used by computer algebra specialists. Using simulation platforms may provide preliminary conclusions but they have to be combined results obtained based on real life environments.

Bibliography

[1] Web Services Description Language (WSDL) 2.0, The World Wide Web Consortium (W3C), http://www.w3.org/TR/wsdl20/, 2007.

[2] A Framework for Brokering Distributed Mathematical Services (MathBroker), http://www.risc.jku.at/projects/mathbroker/, 2011.

[3] GAP - Groups, Algorithms, and Programming, Version 4.4.12, The GAP Group, http://www.gap-system.org, 2011.

[4] Generating a Java Client Proxy and a Sample Application from a WSDL Document using the Axis run-time Environment, http://publib.boulder.ibm.com/infocenter/radhelp/v8r5/index.jsp?topic=%2Forg.eclipse.jst.ws.axis.ui.doc.user%2Ftopics%2Ftsampappa.html. Technical report, Eclipse IBM, 2011.

[5] gLite Grid Middleware, CERN, http://glite.cern.ch, 2011.

[6] Grid Computing Toolbox, Maplesoft, http://www.maplesoft.com/products/toolboxes/gridcomputing/index.aspx, 2011.

[7] Grid-Enabled Numerical and Symbolic Services(GENSS), http://genss.cs.bath.ac.uk/, 2011.

[8] gridMathematica, Wolfram Research, http://www.wolfram.com/gridmathematica/, 2011.

[9] Java Remote Method Invocation (RMI), http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html, 2011.

[10] Maple Computer Algebra System, Maplesoft Inc., http://www.maplesoft.com/, 2011.

[11] Mathematics on the Net (MONET) Consortium, http://monet.nag.co.uk/monet/, 2011.

[12] Maxima, http://maxima.sourceforge.net/, 2011.

[13] MuPad, SciFace Software Gmbh, http://www.sciface.com, 2011.

[14] QMath, http://www.matracas.org/qmath/, 2011.

[15] REDUCE, http://reduce-algebra.com/, 2011.

[16] SCSCP C/C++ Package, http://www.imcce.fr/Equipes/ASD/trip/scscp/, 2011.

[17] SCSCP Java Package, http://java.symcomp.org/, 2011.

[18] Sentido, http://www.matracas.org/sentido/, 2011.

[19] Symbolic Computation Infrastructure for Europe (SCIEnce), http://www.symbolic-computing.org/, 2011.

[20] Systinet Developer for Eclipse, Sysnet, http://www.systinet.com/, 2011.

[21] The Kant Group, Technical University of Berlin, http://page.math.tu-berlin.de/~kant/, 2011.

[22] Universal Description Discovery and Integration (UDDI), https://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm, 2011.

[23] WebServiceStudio 2.0, http://www.gotdotnet.com/team/, 2011.

[24] Wolfram Mathematica, Wolfram Research, http://www.wolfram.com/, 2011.

[25] WSFL, IBM, http://xml.coverpages.org/wsfl.html, 2011.

[26] Axiom Computer Algebra System, http://axiom-developer.org/, 2012.

[27] W. v. D. Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. Technical report, Queensland University of Technology, Brisbane, 2002.

[28] Active Endpoints. ActiveBPEL Designer, http://www.active-endpoints.com/active-bpel-engine-overview.htm, 2011.

[29] C. Adams, S. Farrell, and T. Kause. Internet X. 509 public key infrastructure certificate management protocol (CMP). *Internet Engineering Task Force*, pages 1–96, 2005.

[30] M. Ahsant, M. Surridge, T. Leonard, A. Krishna, and O. Mulmo. Dynamic Trust Federation in Grids. In *Proceedings of the 4th international conference on Trust Management*, number 511563, pages 3–18, Pisa, Italy, 2006. Springer.

[31] M. Aird, W. Medina, J. Davenport, and J. Padget. Description and generation of mathematical web services. In *(e-Proceedings Internet Accessible Mathematical Computation) IAMC 2004*, Santander Spain, 2004.

[32] M. Aird, W. Medina, and J. Padget. MONET: service discovery and composition for mathematical problems. *CCGrid 2003. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003. Proceedings.*, pages 678–685, 2003.

[33] I. a. Ajwa. A case study of Grid Computing and computer algebra: parallel Gröbner Bases and Characteristic Sets. *The Journal of Supercomputing*, 41(1):53–62, Mar. 2007.

[34] A. Akram, D. Meredith, and R. Allan. Evaluation of BPEL to scientific workflows. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 269–274. IEEE, 2006.

[35] M. Aldinucci, M. Danelutto, A. Paternesi, R. Ravazzolo, and M. Vanneschi. Building interoperable grid-aware assist applications via web services. In *Parallel Computing Conference*, volume 33, 2005.

[36] A. A. Almonaies, J. R. Cordy, and R. T. Dean. Legacy system evolution towards service-oriented architecture. In *International Workshop on SOA Migration and Evolution (SOAME10)*, pages 53–62. IEEE Computer Society, 2010.

[37] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guizar, N. Kartha, and Others. Web services business process execution language version 2.0. Technical Report April, OASIS, 2007.

[38] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, third edition, 1999.

[39] A. Ankolekar, M. Burstein, and J. Hobbs. DAML-S: Semantic markup for web services. In *Proceedings Semantic Web Working Symposium*, pages 411–430. Stanford University, Stanford, 2001.

[40] Apache Software Foundation. Apache Axis, http://axis.apache.org/axis/, 2011.

[41] L. Aversano, G. Canfora, A. Cimitile, and A. De Lucia. Migrating legacy systems to the Web: an experience report. *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, pages 148–157, 2001.

[42] R. Baraka, O. Caprotti, and W. Schreiner. Publishing and Discovering Mathematical Service Descriptions: A Web Registry Approach. Technical report, Technical report, RISC-Linz Technical Report, 2004.

[43] R. Baraka and W. Schreiner. Querying registry-published mathematical Web Services. *20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06)*, pages 767–772, 2006.

[44] A. Barros, M. Dumas, and A. ter Hofstede. Service interaction patterns: Towards a reference framework for service-based business process interconnection. Technical report, Faculty of IT, Queensland University of Technology, Queensland, 2005.

[45] J. Basney, M. Humphrey, and V. Welch. The MyProxy online credential repository. *Software: Practice and Experience*, 35(9):801–816, July 2005.

[46] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[47] A. Bayucan, R. L. Henderson, C. Lesiak, B. Mann, T. Proett, and D. Tweten. Portable batch system: External reference specification. Technical report, MRJ Technology Solutions, 1999.

[48] W. H. Bell, D. G. Cameron, A. P. Millar, L. Capozza, K. Stockinger, and F. Zini. Optorsim: A grid simulator for studying dynamic data replication strategies. *International Journal of High Performance Computing Applications*, 17(4):403–416, 2003.

[49] A. Boyle and B. Caviness. Future directions for research in symbolic computation. In A. Boyle and B. Caviness, editors, *Report of a Workshop on Symbolic and Algebraic Computation*, page 95, 1990.

[50] R. Brent. Some parallel algorithms for integer factorisation. In *Euro-Par'99 Parallel Processing*, pages 1–22, London, 1999. Springer-Verlag London.

[51] S. Buswell, O. Caprotti, and M. Dewar. Mathematical Service Description Language: Final Version. Technical report, Monet Consortium, 2003.

[52] G. Butler. Software architectures for computer algebra: a case study. In *Design and Implementation of Symbolic Computation Systems (DISCO 96)*, pages 277–286. Springer-Verlag London, 1996.

[53] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. In *High Performance Computing in the Asia-Pacific Region*, pages 283 – 289. IEEE Computer Society, 2000.

[54] R. Buyya and M. Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220, 2002.

[55] G. Canfora, A. Fasolino, G. Frattolillo, and P. Tramontana. Migrating interactive legacy systems to Web services. *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 10 – 36, 2006.

[56] L. Canon, O. Dubuisson, J. Gustedt, and E. Jeannot. Defining and controlling the heterogeneity of a cluster: The Wrekavoc tool. *Journal of Systems and Software*, 83(5):786–802, 2010.

[57] O. Caprotti and W. Schreiner. MathBroker Overview. Technical report, Johannes Kepler University, Linz, 2002.

[58] A. Carstea, M. E. Frincu, A. Konovalov, G. Macariu, and D. Petcu. On service-oriented symbolic computing. *Parallel Processing and Applied Mathematics*, pages 843–851, 2007.

[59] A. Carstea, M. E. Frincu, G. Macariu, and D. Petcu. Event-Based Simulator for SymGrid-Services Framework. *International Journal of Grid and Utility Computing*, 2(1):33–44, 2011.

[60] A. Carstea, M. E. Frincu, G. Macariu, D. Petcu, and K. Hammond. Generic Access to Web and Grid-based Symbolic Computing Services: the SymGrid-Services Framework. *Sixth International Symposium on Parallel and Distributed Computing (ISPDC'07)*, pages 22–22, July 2007.

[61] A. Carstea and G. Macariu. Towards a grid enabled symbolic computation architecture. *Pollack Periodica*, 3(2):15–26, 2008.

[62] A. Cârstea, G. Macariu, M. Frincu, and D. Petcu. Composing Web-based mathematical services. In *Symbolic and Numeric Algorithms for Scientific Computing, 2007. SYNASC. International Symposium on*, pages 327–334. IEEE, 2007.

[63] A. Carstea, G. Macariu, M. E. Frincu, and D. Petcu. Secure Orchestration of Symbolic Grid Services. In *Proceedings of High Performance Grid Middleware (HiperGRID 2008)*, pages 25–33, Bucharest, 2008. Politehnica Press - IEEE Romania Section.

[64] A. Carstea, G. Macariu, M. E. Frincu, and D. Petcu. Workflow Management for Symbolic Grid Services. *2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 373–379, 2008.

[65] A. Carstea, G. Macariu, M. E. Frincu, and D. Petcu. Description and Execution of Patterns for Symbolic Computations. *2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 197–204, Sept. 2009.

[66] A. Carstea, G. Macariu, D. Petcu, and A. Konovalov. Pattern based composition of Web services for symbolic computations. *Computational ScienceICCS 2008*, pages 126–135, 2008.

[67] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a generic framework for large-scale distributed experiments. In *Computer Modeling and Simulation, 2008. UKSIM 2008. Tenth International Conference on*, pages 126–131. IEEE, 2008.

[68] A. Chakrabarti. Taxonomy of Grid Security Issues. In *Grid Computing Security*, pages 33–47. Springer, 2007.

[69] D. Chicha, Yannis Riem, Manfred Roberts. The MONET Broker. Deliverable D16-D18. Technical report, The MONET Consortium, 2004.

[70] M. Chowdhury and M. Iqbal. Integration of legacy systems in software architecture. *SAVCBS 2004 Specification and Verification of Component-Based Systems*, page 110, 2004.

[71] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming scientific and distributed workflow with Triana services. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, Aug. 2006.

[72] F. Clemente, A. Ortega, and J. Blaya. Distributed Provision and Management of Security Services in Globus Toolkit 4. *Grid Computing, High Performance and Distributed Applications (GADA) 2006 International Conference*, pages 1325–1335, 2006.

[73] S. Comella-Dorda, K. Wallnau, R. Seacord, and J. Robert. A survey of legacy system modernization approaches. Technical Report April, Software engineering institute, Carnegie Mellon, 2000.

[74] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger. Workflow management in Condor. In *Workflows for e-Science*, pages 357–375. Springer Verlag, 1 edition, 2007.

[75] A. De Lucia, G. Di Lucca, A. Fasolino, P. Guerra, and S. Petruzzelli. Migrating legacy systems towards object-oriented platforms. *Proceedings International Conference on Software Maintenance*, pages 122–129, 1997.

[76] E. Deelman and Y. Gil. Managing Large-Scale Scientific Workflows in Distributed Environments: Experiences and Challenges. *2006 Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*, pages 144–144, Dec. 2006.

[77] E. Deelman, M. Livny, G. Mehta, A. Pavlo, G. Singh, M. Su, K. Vahi, and R. Wenger. Pegasus and DAGMan From Concept to Execution : Mapping Scientific Workflows onto Today's Cyberinfrastructure. In *High Performance Computing and Grids in Action*, pages 56–74. IOS, Amsterdam, 2008.

[78] M. Diab. Systolic architectures for multiplication over finite field GF(mˆ2). In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes (AAECC-8)*, pages 329–340. Springer-Verlag, 1991.

[79] E. Dolan, P. Hovland, J. Moré, B. Norris, and B. Smith. Remote Access to Mathematical Software. Technical report, Argonne National Laboratory, Argonne, 2001.

[80] J. Dongarra. NetSolve: A network server for solving computational science problems. *Journal of Supercomputer Applications and High Performance Computing*, 11(3):212—-223, 1997.

[81] T. Dörnemann, T. Friese, S. Herdt, and E. Juhnke. Grid workflow modelling using grid-specific BPEL extensions. In *German e-Science Conference*, pages 1–9, 2007.

[82] T. Dornemann, M. Smith, and B. Freisleben. Composition and Execution of Secure Workflows in WSRF-Grids. In *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*, pages 122–129. IEEE, May 2008.

[83] A. Duscher. An Execution Environment for Mathematical Services based on WSRF and WS-BPEL. Technical report, Johannes Kepler University, Linz, 2005.

[84] A. Duscher. Interaction patterns of mathematical services. Technical report, Johannes Kepler University, Linz, Austria, 2006.

[85] S. Dustdar and W. Schreiner. A survey on web services composition. *Journal of Web and Grid Services*, 1(1):1–30, 2005.

[86] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. L. Price. Grid Service Orchestration Using the Business Process Execution Language (BPEL). *Journal of Grid Computing*, 3(3-4):283–304, Jan. 2006.

[87] P. Enslow. What is a "Distributed" Data Processing System? *Computer*, 11(1):13–21, Jan. 1978.

[88] R. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[89] J. Fischer, A. Schreiber, and M. Strietzel. Script Wrapper for Software Integration Systems. In *High Performance Computing and Networking*, pages 560–563. Springer, 2000.

[90] I. Foster. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, Aug. 2001.

[91] I. Foster. Globus toolkit version 4: Software for service-oriented systems. *Network and Parallel Computing*, pages 2–13, 2005.

[92] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure WG, Global Grid Forum*, volume 22, pages 1–5. Edinburgh, 2002.

[93] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The physiology of the grid. In G. F. T. H. Fran Berman, editor, *Grid Computing: Making the Global Infrastructure a Reality*, pages 217–249. John Wiley and Sons Inc, 2003.

[94] A. Franke and M. Kohlhase. System description: MathWeb, an agent-based communication layer for distributed automated theorem proving. *Automated DeductionCADE-16*, pages 676–676, 1999.

[95] S. Freundt, P. Horn, A. Konovalov, and S. Linton. Symbolic computation software composability. *Intelligent Computer Mathematics*, page 285, 2010.

[96] S. Freundt, P. Horn, A. Konovalov, S. Linton, and D. Roozemond. Symbolic Computation Software Composability Protocol (SCSCP) specification. Technical report, SCIEnce, 2008.

[97] J. Frey, T. Tannenbaum, M. Livny, and I. Foster. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, pages 55–63, 2002.

[98] M. E. Frincu, G. Macariu, and A. Carstea. Dynamic and adaptive workflow execution platform for symbolic computations. *Pollack Periodica*, 4(1):145–156, Apr. 2009.

[99] D. Gannon, J. Alameda, O. Chipara, M. Christie, V. Dukle, L. Fang, M. Farrellee, G. Kandaswamy, D. Kodeboyina, S. Krishnan, and Others. Building grid portal applications from a web service component architecture. In *Proceedings of the IEEE*, volume 93, pages 551–563. IEEE, 2005.

[100] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.

[101] D. Garlan and J. M. Ockerbloom. Architectural Mismatch : Why Reuse is Still So Hard. *IEEE Software*, 25(4):66–69, 2009.

[102] D. Garlan and M. Shaw. An introduction to software architecture. *Advances in software engineering and knowledge engineering*, 1(January):1–40, 1993.

[103] M. Gastineau and J. Laskar. TRIP 1.2.0, http://www.imcce.fr/trip/.

[104] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[105] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the challenges of scientific workflows. *Computer*, 40(12):24–32, Feb. 2007.

[106] T. Glatard, D. Emsellem, and J. Montagnat. Generic web service wrapper for efficient embedding of legacy codes in service-based workflows. In *Grid-Enabling*

Legacy Applications and Supporting End Users Workshop (GELA06),, Paris, France, 2006. IEEE Computer Society.

[107] J. Grabmeier, E. Kaltofen, and V. Weispfenning. *Computer Algebra Handbook*. Springer-Verlag New York, Inc., 2003.

[108] N. Gray. Comparison of Web Services, Java-RMI, and CORBA service implementations. In *The Fifth Australasian Workshop on Software and System Architectures (AWSA 2004)*, page 52. Australian Computer Society, 2004.

[109] D. R. Grayson and M. E. Stillman. Macaulay2, a software system for research in algebraic geometry, http://www.math.uiuc.edu/Macaulay2/, 2012.

[110] K. Hammond, A. Zain, G. Cooperman, D. Petcu, and P. Trinder. SymGrid: a framework for symbolic computation on the Grid. *Euro-Par 2007 Parallel Processing*, pages 457–466, Sept. 2007.

[111] Y. Huang, I. Taylor, D. Walker, and R. Davies. Wrapping legacy codes for grid-based applications. In *Proceedings International Parallel and Distributed Processing Symposium*, page 7. IEEE Computer Society, 2003.

[112] Y. Huang and D. Walker. JACAW: A Java-C Automatic Wrapper. Technical report, Cardiff University, Wales, UK, 2002.

[113] T. Jebelean. Integer and rational arithmetic on MasPar. *Design and Implementation of Symbolic Computation*, LNCS 1128:162–173, 1996.

[114] T. Jebelean, M. Dragan, D. Tepeneu, and V. Negru. Parallel Algorithms for Practical Multiprecision Arithmetic Using the Karatsuba Method. Technical report, RISC, Linz, Austria, 2000.

[115] P. Kacsuk and G. Sipos. Multi-Grid, Multi-User Workflows in the P-GRADE Grid Portal. *Journal of Grid Computing*, 3(3-4):221–238, Jan. 2006.

[116] B. Kiepuszewski. *Expressiveness and suitability of languages for control flow modelling in workflows*. PhD thesis, Queensland University of Technology, 2002.

[117] B. Kiepuszewski, A. ter Hofstede, and C. Bussler. On structured workflow modelling. In *Advanced Information Systems Engineering*, pages 431–445. Springer, 2000.

[118] M. Kohlhase. OMDoc: An infrastructure for OpenMath content dictionary information. *ACM SIGSAM Bulletin*, 34(2):43–48, 2000.

[119] M. Kohlhase. Omdoc: Towards an internet standard for the administration, distribution, and teaching of mathematical knowledge. *Artificial Intelligence and Symbolic Computation*, pages 32–52, 2001.

[120] H. Lamehamedi, Z. Shentu, B. Szymanski, and E. Deelman. Simulation of dynamic data replication strategies in data grids. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 10. IEEE Computer Society, 2003.

[121] G. Laszewski and M. Hategan. Workflow Concepts of the Java CoG Kit. *Journal of Grid Computing*, 3(3-4):239–258, Jan. 2006.

[122] K. Leai, L. Tan, and K. J. Turner. Automated Analysis and Implementation of Composed Grid Services. *South-East European Workshop on Formal Methods*, (November):51–64, 2007.

[123] A. Leykin. On parallel computation of Gröbner bases. In *ICPP Workshops*, pages 160–164. IEEE Computer Society, 2004.

[124] S. Linton and A. Solomon. OpenMath, IAMC and GAP. In *Proceedings of ISSAC 99/IAMC workshop*, pages 1–14. ACM SIGSAM/SIGNUM, 1999.

[125] F. Lu, H. Huang, Z. Xu, and H. Yu. A Middleware for legacy application wrapper. *First International Conference on Semantics, Knowledge and Grid*, (Skg 2005):47–47, Dec. 2005.

[126] F. Lubeck and M. Neunhoffer. Enumerating large orbits and direct condensation. *Experimental Mathematics*, 10(2):197–206, 2001.

[127] S. Ludwig, W. Naylor, J. Padget, and O. Rana. Matchmaking support for mathematical web services. In *Proceedings of the UK e-science all hands meeting*, number 1, 2005.

[128] S. Ludwig, O. Rana, W. Naylor, and J. Padget. Matchmaking Portal for the Discovery of Numerical and Symbolic Services. In *Proceedings of 4th UK e-Science Programme All Hands Meeting (AHM)*, Nottingham, UK, 2005.

[129] G. Macariu, A. Cârstea, M. E. Fr\ˆ\incu, and D. Petcu. Towards a Grid Oriented Architecture for Symbolic Computing. *2008 International Symposium on Parallel and Distributed Computing*, pages 259–266, 2008.

[130] S. Majithia, M. Shields, I. Taylor, and I. Wang. Triana: a graphical Web service composition and execution toolkit. In *Proceedings. IEEE International Conference on Web Services, 2004.*, pages 514–521. IEEE Computer Society, 2004.

[131] M. Matooane. Parallel systems in symbolic and algebraic computation. Technical Report 537, University of Cambridge, Cambridge, 2002.

[132] J. McCall, P. Richards, and G. Walters. Factors in software quality. Technical Report November, General Electric Company, 1977.

[133] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3/4):623, 1994.

[134] B. Michael, G. Yaron, K. Matthias, L. Frank, P. Gerhard, R. Dieter, and R. Michael. BPELJ : BPEL for Java Authors. Technical Report March, BEA Systems Inc and IBM Corporation, 2004.

[135] N. Milanovic and M. Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004.

[136] M. Morii and Y. Takamatsu. Exponentiation in finite fields using dual basis multiplier. *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, 508/1991:354–366, 1991.

[137] W. Naylor and J. Padget. Semantic matching for mathematical services. In *Mathematical Knowledge Management*, pages 174–189. Springer, 2006.

[138] C. Neuman, T. Yu, and S. Hartman. The Kerberos Network Authentication Service (V5), July 2005. *URL ftp://ftp. isi. edu/in-notes/rfc1510. txt. RFC*, (July):1–119, 2005.

[139] R. Nick, A. ter Hofstede, W. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical report, BPM Center Report, 2006.

[140] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, Aug. 2006.

[141] Organization for the Advancement of Structured Information Standards (OASIS). WS-BaseFaults Standard, http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-BaseFaults-1.2-draft-02.pdf, 2011.

[142] Organization for the Advancement of Structured Information Standards (OASIS). WS-Notification Standard, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn, 2011.

[143] Organization for the Advancement of Structured Information Standards (OASIS). WS-Resource Standard, http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf, 2011.

[144] Organization for the Advancement of Structured Information Standards (OASIS). WS-ResourceLifetime Standard, http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-03.pdf, 2011.

[145] Organization for the Advancement of Structured Information Standards (OASIS). WS-ServiceGroup Standard, http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ServiceGroup-1.2-draft-02.pdf, 2011.

[146] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, Oct. 2003.

[147] D. Petcu. Between Web and Grid-based Mathematical Services. In *Computing in the Global Information Technology, 2006. ICCGI'06. International Multi-Conference on*, page 41. IEEE, 2007.

[148] D. Petcu, A. Carstea, G. Macariu, and M. E. Frincu. Service-oriented Symbolic Computing with SymGrid. *Scalable Computing: Practice and Experience*, 9(2):111–124, 2008.

[149] D. Petcu and D. Dubu. An Extension of Maple for Grid and Cluster Computing. *Studies in Informatics and Control*, 14(1):31, 2004.

[150] D. Petcu, G. Macariu, A. Carstea, and M. E. Frincu. Service-Oriented Symbolic Computations. In *Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications*, pages 1053–1075. Information Science Publishing, 2010.

[151] D. Petcu, M. Paprzycki, and D. Dubu. Design and implementation of a grid extension for Maple. *Scientific Programming*, 13(2):137–149, 2005.

[152] D. Petcu, D. Tepeneu, M. Paprzycki, and T. Ida. Symbolic computations on Grids. *Engineering the Grid: status and perspective*, pages 1–17, 2006.

[153] D. Petcu, D. Tepeneu, M. Paprzycki, T. Mizutani, and T. Ida. Survey of Symbolic Computations on the Grid. In *3rd International Conference: Science of Electronic Technologies of Information and Telecommunications, Tunisia*. IEEE Computer Society, 2005.

[154] K. Qian, X. Fu, and L. Tao. *Software architecture and design illuminated*. Jones & Bartlett Publishers, 2009.

[155] J. Ransom, I. Somerville, and I. Warren. A method for assessing legacy systems for evolution. In *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*, pages 128–134. IEEE Comput. Soc, 1998.

[156] J. Rao. A survey of automated web service composition methods. In *Semantic Web Services and Web Process Composition*, volume LNCS 3387/, pages 43–54. Springer Verlag, 2005.

[157] M. Riem. The OpenMath Guide. Technical report, RIACA, 2004.

[158] Satish Thatte. XLANG, Web Services for Business Process Design. Technical report, Microsoft Corporation, 2001.

[159] W. Schreiner. A Distributed Computer Algebra System Based on Maple and Java. Technical Report RISC Report Series, University of Linz, Schloss Hagenberg, Linz, Austria, 1999.

[160] W. Schreiner, C. Mittermaier, and K. Bosa. Distributed Maple: Parallel computer algebra in networked environments. *Journal of Symbolic Computation*, 35(3):305–347, 2003.

[161] M. Senger, P. Rice, and T. Oinn. Soaplab-a unified Sesame door to analysis tools. In *Proceedings of the UK e-Science All Hands Meeting*, volume 18, pages 509–513, 2003.

[162] K. Seymour, A. YarKhan, S. Agrawal, and J. Dongarra. Netsolve: Grid enabling scientific computing environments. *Advances in Parallel Computing*, 14(May 2005):33–51, 2005.

[163] M. Shaw. Architectural issues in software reuse: it's not just the functionality, it's the packaging. *Journal of Parallel and Distributed Computing*, 59(2):107 – 131, 1999.

[164] M. Shoukat, M. Maheswaran, and S. Ali. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Distributed Computing*, 1999.

[165] E. Sibert, H. Mattson, and P. Jackson. Finite field arithmetic using the connection machine. *Computer Algebra and Parallelism*, pages 51–61, 1992.

[166] A. Slominski. Adapting BPEL to scientific workflows. In *Workflows for e-Science*, pages 208–226. 2007.

[167] H. Sneed. Encapsulating legacy software for use in client/server systems. *Proceedings of WCRE '96: 4rd Working Conference on Reverse Engineering*, pages 104–119, 1996.

[168] H. Sneed. Wrapping legacy software for reuse in a SOA. In *Multikonferenz Wirtschaftsinformatik*, volume 2, pages 345–360. GITO mbH Verlag, 2006.

[169] A. Solomon. Distributed computing for conglomerate mathematical systems. *Algebra, Geometry and Software System*, pages 309–325, 2007.

[170] A. Solomon and C. Struble. JavaMath: an API for internet accessible mathematical services. In *Computer mathematics: proceedings of the fifth Asian Symposium (ASCM 2001), Matsuyama, Japan, 26-28 September 2001*, pages 151–160. World Scientific Pub Co Inc, 2001.

[171] B. Srivastava and J. Koehler. Web service composition-current solutions and open problems. In *ICAPS 2003 Workshop on Planning for Web Services*, volume 35, pages 28 – 35. AAAI Press, 2003.

[172] H. Stockinger. Grid Computing: A Critical Discussion on Business Applicability. *IEEE Distributed Systems Online*, 7(6):2–2, June 2006.

[173] A. Takefusa, S. Matsuoka, H. Nakada, K. Aida, and U. Nagashima. Overview of a performance evaluation system for global computing scheduling algorithms. In

High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on, pages 97–104. IEEE, 1999.

[174] K. L. L. Tan and K. J. Turner. Orchestrating grid services using BPEL and Globus Toolkit 4. In *Proc. 7th PGNet Symposium*, pages 31–36. School of Computing, Liverpool John Moores University, 2006.

[175] A. S. Tanenbaum and M. V. Steen. *Distributed Systems Principles and Paradigms*. Prentice Hall, 2003.

[176] B. H. Tay and A. L. Ananda. A Survey of Remote Procedure Call. *ACM SIGOPS Operating Systems*, 24(3):68–79, July 1990.

[177] I. Taylor, M. Shields, I. Wang, and A. Harrison. Visual Grid Workflow in Triana. *Journal of Grid Computing*, 3(3-4):153–169, Jan. 2006.

[178] D. Tepeneu and T. Ida. MathGridLink-A bridge between Mathematica and the Grid. *Proc. JSSST*, 3:74–77, 2003.

[179] D. Thain and T. Tannenbaum. Distributed computing in practice: The Condor experience. *Practice and Experience*, 17(2-4):323–356, 2005.

[180] The MONET Consortium. MONET Architecture Overview. Deliverable D04. Technical report, 2003.

[181] The Novell Corporation. Novell exteNd Workbench, http://www.novell.com/developer/ndk/extend.html.

[182] The Numerical Algorithms Group. NAG Numerical Library, http://www.nag.com/numeric/numerical_libraries.asp, 2011.

[183] The Object Management Group (OMG). Common Object Request Broker(CORBA), http://www.omg.org/spec/CORBA/3.1.1/, 2011.

[184] The OpenMath Society. OpenMath, http://www.openmath.org/, 2011.

[185] The Progress Software Corporation. Stylus Studio, http://www.stylusstudio.com, 2012.

[186] The World Wide Web Consortium(W3C). Mathematical Markup Language (MathML) , http://www.w3.org/TR/MathML3/chapter4.html#contm.strict, 2011.

[187] The World Wide Web Consortium(W3C). Mathematical Markup Language (MathML), http://www.w3.org/TR/MathML3/, 2011.

[188] The World Wide Web Consortium(W3C). WS-ADDRESSING, http://www.w3.org/Submission/ws-addressing/, 2011.

[189] K. J. Turner. Representing and analysing composed web services using Cress. *Journal of Network and Computer Applications*, 30(2):541–562, Apr. 2007.

[190] K. J. Turner and K. Tan. Graphical composition of grid services. In *Proceedings of the 3rd international conference on Rapid integration of software engineering techniques*, number May, pages 1–17. Springer-Verlag, 2006.

[191] W. van der Aalst. Dont go with the flow: Web services composition standards exposed. *IEEE intelligent systems*, 18(1):72–76, 2003.

[192] W. van Der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003.

[193] D. Wagner. MathLink mode. *The Mathematica Journal*, 1996.

[194] B. Wassermann, W. Emmerich, B. Butchart, N. Cameron, L. Chen, and J. Patel. Sedna: A BPEL-based environment for visual scientific workflow modeling. *Workflows for e-Science*, pages 428–449, 2007.

[195] S. Watt. *Bounded parallelism in computer algebra*. PhD thesis, Waterloo, 1986.

[196] S. Watt. On the future of Computer Algebra Systems at the threshold of 2010. *Proceedings ASCM-MACIS*, pages 422–430, 2009.

[197] A. Weber, W. Küchlin, and B. Eggers. Parallel computer algebra software as a Web component. *Concurrency: Practice and Experience*, 10(11-13):1179–1188, Sept. 1998.

[198] P. Wohed, W. v. D. Aalst, M. Dumas, and A. ter Hofstede. Analysis of web services composition languages: The case of BPEL4WS. *Conceptual Modeling-ER 2003*, pages 200–215, 2003.

[199] J. Yu and R. Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, 3(3-4):171–200, Jan. 2006.

[200] A. Zain, K. Hammond, and P. Trinder. SymGrid-Par: Designing a framework for executing computational algebra systems on Computational Grids. In *Intl. Conference on Computer Science (ICCS)*, pages 617–624, Beijing, 2007. Springer Verlag.

[201] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, Reliable, Loosely Coupled Parallel Computation. *2007 IEEE Congress on Services (Services 2007)*, pages 199–206, July 2007.